A Programming Environment for Reactive and Concurrent Systems Using Petri Nets and Temporal Logic

ペトリネットと時相論理を用いた
リアクティブ並行システムの開発環境の研究

Naoshi Uchihira

内平 直志

Graduate School of Information Science and Engineering
Tokyo Institute of Technology

東京工業大学 情報理工学研究科 計算工学専攻

A thesis submitted to Tokyo Institute of Technology
in partial fulfillment of the requirement for
the degree of Doctor of Engineering

December, 1997

ABSTRACT

There has been a rapid trend towards parallel, distributed, and interactive/reactive computing over the past decade. Generally speaking, it is not so easy for ordinary programmers to produce correct and efficient programs for these systems as compared with sequential programming. Therefore, some kind of computer-aided concurrent programming environment is necessary to achieve high productivity and high reliability. The purpose of this thesis is to present theories, methods and tools (programming environments) for reactive and concurrent systems using Petri nets and temporal logic.

Both Petri nets and temporal logic have been investigated as formal specification languages for reactive and concurrent systems. While temporal logic is appropriate for specifying the properties and constraints of programs but inappropriate for specifying the behavioral structures of programs, Petri nets can specify the behavioral structures but not the properties and constraints. In this thesis, the fusion of Petri nets and temporal logic is proposed as a specification language for reactive and concurrent systems. Then, practical and efficient verification and synthesis methods using Petri nets and temporal logic and Petri-net-based design methodology are described. Finally, a programming environment embedding these methods is introduced. This thesis attempts in illustrating a typical programming paradigm and its environment using Petri nets and temporal logic.

The main outcomes of this thesis are as follows.

(1) Specification by fusion of Petri nets and temporal logic

The fusion of Petri nets and temporal logic is proposed as a specification language, and its applications to verification and synthesis are considered. The remarkable point is that the proposed methods are for unbounded Petri nets, while former verification and synthesis methods were mainly for bounded (i.e., finite) ones.

(2) Compositional verification

An efficient and practical verification method using transition systems (bounded Petri nets) and temporal logic is proposed. Generally, the computation costs for verification increase exponentially as the scale of the programs increases. To overcome this problem, a reduction technique of the target program has been investigated using bisimulation equivalence. However, bisimulation equivalence cannot deal with "divergence" explicitly. Therefore, a new process equivalence relation ($\pi\tau\omega$-bisimulation equivalence) is proposed. A Process Query Language (PQL) which is an extended temporal logic and semantics of which is based on $\pi\tau\omega$-bisimulation equivalence is defined. Then, a compositional verification method using PQL is proposed.

(3) Compositional program adjustment

A new synthesis method using transition systems (bounded Petri nets) and temporal logic is proposed. Since conventional program generation from a temporal logic specification is impractical, this thesis proposes a new approach, "program adjustment". In program adjustment, a target program written by programmers which may be functionally correct but may be imperfect in its timing is automatically adjusted (tuned up) to satisfy given temporal logic constraints.

(4) Petri-net-based software design methodology

A Petri-net-based software design method is proposed. In this method, a causality matrix is introduced for an earlier design phase when the system structure is obscure and it is difficult to write Petri nets directly. A designer can construct Petri nets systematically from an ambiguous requirement using the causality matrices according to the design method.

(5) MENDELS ZONE

A programming environment, MENDELS ZONE, based on the above techniques has been developed. MENDEL net which is a high-level Petri net for reactive and concurrent systems is used as the programming language. The designer constructs a program (MENDEL net) and verifies it using temporal logic. If there are any bugs, the program can be adjusted. Finally, the constructed program is executed on a parallel computer.

概要

計算機システムの並列化，分散化，インタラクティブ/リアクティブ化に伴い，並行プログラミングの需要はますます大きくなりつつある．しかしながら，並行プログラムの開発は，逐次プログラムの開発に比べて格段に難しい．さらに，制御システムなどのリアクティブシステムの並行プログラムには高い信頼性が要求されることが多い．このようなソフトウェアの高生産性および高信頼性を確保するためには，ソフトウェア開発の計算機支援は不可欠である．本研究の目的は，ペトリネットと時相論理を用いたリアクティブ・並行システムのソフトウェア開発支援技術を確立することである．

ペトリネットと時相論理はともに並行システムの形式的な仕様記述法として研究されてきた．しかし，時相論理は制約記述には適しているが構造記述には不適であり，逆にペトリネットは構造記述には適しているが制約記述には不適であった．そこで，本研究では時相論理とペトリネットの融合によるリアクティブ・並行システムの仕様記述法を採用する．さらに，ペトリネットと時相論理を用いた実用的な検証・合成手法などの要素技術を開発し，それらを組み込んだプログラムの開発環境の試作およびソフトウェア設計方法論の提示を行なう．本研究により，ペトリネットと時相論理を用いた一つの具体的なソフトウェア開発支援体系を確立することができた．

本研究の主な具体的成果としては，以下の項目がある．

(1) ペトリネットと時相論理の融合による仕様記述法の提案，

　時相論理とペトリネットの融合法を示し，ペトリネットが時相論理を満たすか否かを検証するアルゴリズムを示した．従来，有界なペトリネットに対する時相論理による検証法は知られていたが，それを有界でない一般のペトリネットの検証に拡張した．

(2) 積み重ね式検証法の研究

　遷移システム (有限ペトリネット) と時相論理を用いた大規模プログラムに対する効率的な検証法を示した．従来，時相論理のモデル検査法では検証に要するコストの爆発的増大が問題あり，これを回避するために様々な方法が提案され，その１つに検証対象の縮約手法がある．しかし，従来の方式では,「内部遷移による無限ループ」の処理に関して問題があった．本研究では，その問題点を指摘するとともに，それを是正する新しい様相論理 PQL を提案し，PQL を用いた積み重ね検証法を示した．

(3) プログラム調整法の研究

　遷移システム (有限ペトリネット) と時相論理による並行プログラムの新しい合成法を示した．従来の時相論理からのプログラム合成法は，実用規模のプログラムに適用するには仕様記述能力および計算量の点で非現実的であった．そこで，プログラム全体の自動合成ではなく，人間が作成した不完全なプログラム (遷移システム) を時相論理式で記述された仕様を満たすように部分的に自動修正し，目的のプログラムを生成する「プログラム調整法」を提案した．

(4) ネット指向ソフトウェア設計法の提案

　ペトリネットは単なる記述手段であり，あいまいな仕様からペトリネットを導出するためのガイドライン (設計法) が必要である．本研究では，あいまいな仕様をペトリネットに具体化する手段として因果関係マトリックスを導入したリアクティブ・並行システムのソフトウェア設計法を提案した．

(5) プログラミング環境 MENDELS ZONE の開発

　上記の要素技術を統合したプログランミング環境を並列計算機上に試作した．ここで，プログラムは高水準ペトリネットである MENDEL ネットで記述する．設計者はネット指向設計法によりプログラム (MENDEL ネット) を作成し，プログラム検証でプログラムを検証し，もし問題点がある場合はプログラム調整によりプログラムを改善する．さらに，生成されたプログラムを実際に並列マシン上で実行できる．

Acknowledgments

This thesis was supervised by Prof. Motoshi Saeki. I am grateful for his kind and helpful guidance and continuous encouragement which lead me to accomplishing my thesis. Also, this thesis was advised by Prof. Takuya Katayama, Prof. Naoki Yonezaki, Prof. Takehiro Tokuda, Prof. Tomohiro Yoneda, and Prof. Takashi Imaizumi. I am very thankful for their advice.

The greater part of my research has been carried out as a part of MENDELS ZONE project at Toshiba Corporation. My special thanks go to Dr. Shinichi Honiden who was a project leader. At the beginning of the project, he provided me chance to study temporal logic and Petri nets from a software engineering viewpoint. Throughout the project, he has properly supported, guided, and encouraged my research activities.

Hideji Kawata is my research partner who made considerable contribution on the implementation and improvement of our tools, MENDELS ZONE and SAVE/SFC. I wish to express my gratitude for his generous collaboration.

I also have many thanks to the all other members of the MENDELS ZONE project and SAVE/SFC project, especially to Akihiko Ohsuga, Kazunori Matsumoto, Mikako Arami, Toshiaki Kasuya, Toru Uraoka, Akiko Uraoka, and Toshibumi Seki. When PQL was applied to a practical machine control program, the discussion with Satoshi Sumida was very helpful. I would like to thank him.

When joining Toshiba in 1982, I was a novice in software engineering. A lot of senior researchers of our laboratory educate, train, and promote me through the wide range of software engineering activities. These valuable experiences lie at the foundation of this thesis. I would like to thank them, especially Kazuo Matsumura, Hideo Nakamura, Yutaka Ofude, and Shoichi Kojima.

I am also grateful to successive directors and senior managers of the Systems & Software Engineering Laboratory, Seiichi Nishijima, Sadakazu Watanabe, Mutsuhiro Arinobu, Masahiko Arai, Takeshi Kohno, Shinsuke Tamura, Junichiro Tsuda, and Kazuo Mikame. Without their continuous support and encouragement, I could not continue the research about temporal logic and Petri nets during a long term (more than 10 years).

This research has been affected by both Petri net and software engineering communities. Many people gave me fruitful suggestions and critical comments in the related conferences. I would like to tender my thanks to them, especially Prof. Mikio Aoyama and Prof. Kunihiko Hiraishi. The MENDELS ZONE project has been supported by ICOT. I am also indebted to members of ICOT.

Finally, I reserve a special gratitude to my wife, Mari, for mental and physical support. I dedicate this thesis to Mari and our baby girl, Akari.

Contents

List of Practical Examples of Reactive and Concurrent Systems

The following middle-scale examples are relatively practical and used to show that the techniques proposed in this thesis can be applicable to actual systems.

1. Manufacturing machine control system
   compositional verification (Chapter 5)
   transition system + temporal logic (PQL)
2. Chemical plant control system
   compositional verification (Chapter 5)
   Petri net (SFC) + temporal logic (PQL)
3. Lift control system
   Petri-net-oriented design methodology (Chapter 8)
   Petri net (MENDEL net) + temporal logic (PLTL)
4. Power plant control system
   MENDELS ZONE (Chapter 9)
   Petri net (MENDEL net) + temporal logic (PLTL)

Glossary

Notation	**Meaning**
δ	transition function (transition system)
$\perp$	disabled
ρ	transition function (automaton)
τ	unobservable internal action
θ	action/transition sequence (run)
η	label sequence (word)
Δ	deadlock
ε	empty sequence
X^*	a set of finite sequences over X
X^ω	a set of infinite sequences over X
X^∞	$X^\infty = X^* \cup X^\omega$
$\overline{X}$	a complementary set of X
$X_1 \backslash X_2$	$X_1 \backslash X_2 = X_1 \cap \overline{X_2}$
T	a set of transitions
A	a set of actions
Act	$Act = A \cup \{\tau\}$
S	a set of states
Σ	alphabet
π	state attribute assignment function
$TS = (S, P, A, \pi, \delta, s_0)$	transition system
$N = (P, T, w, m_0)$	Petri net
$m_1[\theta > m_2$	m_1 is reachable from m_2 by a transition sequence θ
$\Delta(\theta)$	differential vector
λ	labeling function
$LN = (N, \lambda)$	Labeled Petri net
$P = (S, A, L, \delta, \pi, s_0, F)$	finite state process (FSP)
L/A	sequences of L where all elements except A are deleted
$L(LN)$	Petri net language
$L_\omega(LN)$	Petri net infinite language
$L_{\Delta\omega}(LN)$	Petri net infinite language with deadlock
$L_{\Delta\omega}^{fair}(LN)$	Petri net fair infinite language with deadlock
$TS_1 \mid TS_2$	process composition
$s \uparrow$	divergence
$TS_1 \approx_{\pi\tau\omega} TS_2$	$\pi\tau\omega$-bisimulation equivalence
f	formula
$\Box f$	f is always true.
$\Diamond f$	f will be eventually true.
$\bigcirc f$	f will be true at the next time.
$f_1 \; U \; f_2$	f_1 continues to be true until f_2 becomes true.
$TS \models f$	TS is a model of f
$\ll a \gg^- f$	f will be sometime true after a is occurred (minimum fixed point)
$\ll a \gg^+ f$	f will be sometime true after a is occurred (maximum fixed point)
$[[a]]^- f$	f will be always true after a is occurred (minimum fixed point)
$[[a]]^+ f$	f will be always true after a is occurred (maximum fixed point)

apat f	*f* is true at all time in all paths
apet f	*f* is true at some time in all paths
epat f	*f* is true at all time in some path
epet f	*f* is true at some time in some path

Chapter 1

Introduction

1 Motivation

Computer systems have been moving rapidly toward parallel, distributed, and reactive computing during the past decade. There is an increasing demand for programmers who can design concurrent programs for these systems. However, generally speaking, compared with sequential programming it is not so easy to produce correct and efficient concurrent programs. In particular, the cost of testing and debugging becomes a heavy burden. Moreover, most reactive systems (e.g. embedded control systems, process control systems) require high reliability. Therefore, some kind of computer-aided concurrent programming environments are necessary to enable ordinary programmers to develop concurrent programs while achieving high productivity and high reliability. The programming environment means tools for verification, debugging, performance evaluation, and methods to synthesize correct and efficient programs.

To achieve both high productivity and high reliability, a *formal method* is the most promising approach in the long run. In the formal method, specifications and programs (implementation) are described using some formal language, the semantics of which is formally defined. Then, the formal method provides verification and synthesis methods for these specifications and programs. Many formal methods have been proposed for concurrent systems. For example, they include models of programs (e.g., transition system, Petri net, automaton), logics of programs (e.g., Hoare logic, temporal logic, dynamic logic, process logic), and algebras of programs (e.g., CCS, CSP, ACP). Temporal logic is a especially useful formal framework for specification, verification, and synthesis for concurrent programs. Therefore, the application of temporal logic to computer science has been actively investigated by many researchers all over the world from the late 1970's to the present.

This thesis concentrates on reactive and concurrent systems. The purpose of this thesis is to establish theories, methods and tools (programming environment) for these systems using temporal logic. Especially, much consideration is given to practical techniques in order to apply temporal logic to actual reactive and concurrent systems.

To apply temporal logic to the actual development of reactive and concurrent systems, the following issues should be considered.

1. **Actual systems cannot be fully described by temporal logic**

   Standard temporal logic is too inconvenient to describe actual reactive and concurrent systems. Therefore, a syntactical extension of temporal logic for practical use or combination with other complementary formal languages is required.

2. **Cost of verification and synthesis is beyond practical computing power**

   Generally speaking, computing cost of verification and synthesis based on formal methods becomes very huge due to "state explosion problem". Consequently these methods are often applicable only to "toy programs". Techniques to avoid state explosion and reduce computing cost are required.

3. **Formal methods do not support the whole development process**

   The formal method is not an all-around player. It can support a few parts of the whole development process. Remaining parts of the process are done by traditional and informal methods. There is an obvious gap between formal methods and traditional and informal methods. Therefore, a design methodology (guidelines) is required to specify how to utilize formal methods and informal methods complementarily in the development of reactive and concurrent programs. Furthermore, the programming environment is required to mechanically support the design methodology throughout the entire development process.

This thesis proposes the following solutions for above problems.

1. **Fusion of Petri nets and temporal logic**

   Temporal logic is appropriate for specifying the properties and constraints of programs, but inappropriate for explicitly specifying the behavioral structures of programs. In other words, temporal logic is declarative and not operational. On the other hand, Petri nets can specify the behavioral structures operationally. In this thesis, a fusion of Petri nets and temporal logic is proposed as a specification language for reactive and concurrent systems.

2. **Compositional verification and adjustment**

   A temporal logic model-checking method is actually useful to verify reactive and concurrent systems. However, as the scale of the programs increases, the computation costs for verification increase exponentially. To reduce the computation costs, this thesis introduces a compositional approach into model-checking, and proposes a new modal logic, PQL, and a compositional verification method based on PQL.

   Since it seems that the conventional program generation approach from temporal logic specification is impractical, a new approach "program adjustment" is proposed, in which a target program made by programmers is automatically adjusted (tuned up) to satisfy given temporal logic constraints. Since this also causes the state explosion problem, compositional program adjustment is proposed in this thesis.

3. **Petri-net-based design methodology and its programming environment**

   As a software design methodology for reactive and concurrent systems, several methods have been proposed which include OOAD and RTSAD. However, few are based on Petri nets. Since Petri nets are used as a specification language in our approach, a Petri-net-based design method is proposed which gives guidelines for deriving detailed Petri nets from ambiguous requirements, and then verifying and adjusting programs using Petri nets and temporal logic.

   MENDELS ZONE is a programming environment for reactive and concurrent systems, which supports the software development process from first to last, including verification, adjustment, and Petri-net-based design method.

2 Background

To providing the background for this thesis, this section presents a brief survey of software development techniques for reactive and concurrent systems. Especially, focusing on formal techniques including *specification*, *verification*, *synthesis*, and *design methodology* (Fig. 1).

2.1 Formal Specification for Reactive and Concurrent Systems

Specifying systems formally is an effective way to recognize and clarify ambiguous parts of the system requirement/specification, and is necessary to verify the specification and synthesize programs from the specifications. A lot of specification languages for reactive and concurrent systems have been proposed. Roughly speaking, they can be classified into *operational approaches* and *declarative approaches.*

- **Operational Approach**

  In an operational approach, reactive and concurrent systems are modeled with specification models/languages which are executable on the *abstract machine.* The operational (i.e., executable) specification models for reactive and concurrent systems include *transition systems*, *state machines*, *automata*, *Statechart* and *Petri nets.* These models will be described in detail in Chapter 2. However, these models are too primitive to describe practical systems. Therefore, several practical specification languages which are based on some primitive model and extended from a practical point of

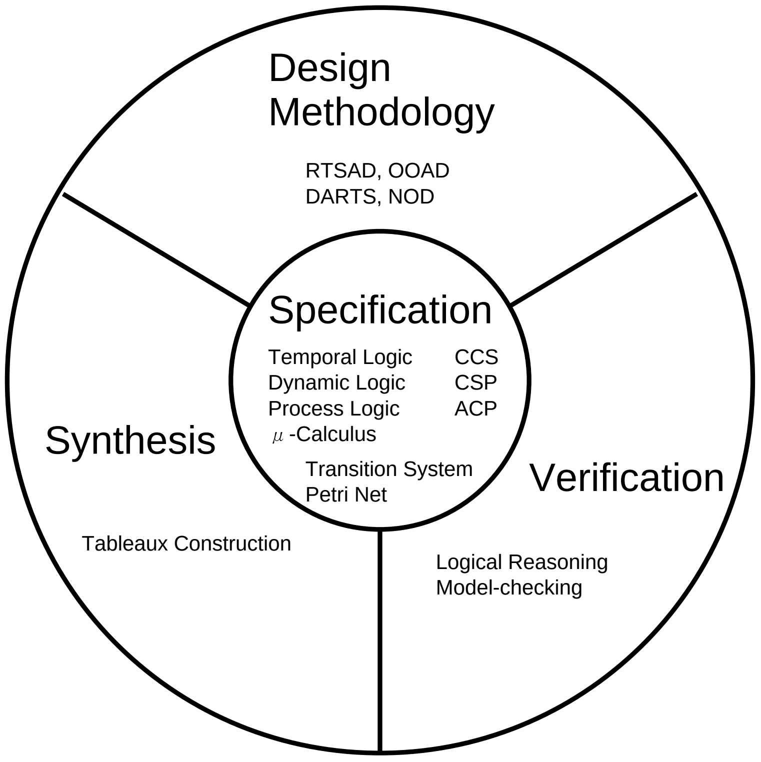


Figure 1. Software Development Techniques for Reactive and Concurrent Systems

view. These specification languages include PAIZLey [Zave 82, Zave 91], SDL, and Coloured Petri Net [Jensen 92, Jensen 95].

- **Declarative Approach**

  In a declarative approach, systems are modeled by logical formulas or algebraic terms, and then axioms of the logics or algebras give the semantics of the systems. For concurrent systems, a lot of declarative formalisms have been proposed, which include modal/program logic (Dynamic Logic, Temporal Logic, Process Logic, Modal μ-calculus, UNITY logic, etc.) and process algebra (CCS, CSP, ACP, π-calculus, etc.).

In this thesis, we focus on *Temporal Logic.* Temporal logic is a kind of modal logic in which modal operators represent the topology of time. While the truth-values of propositions are constant in classical logic, the truth-values of propositions can be changed with time in temporal logic. For example, a statement "if a proposition p is $true$ now, p will have to be $false$ at the next time" can be represented by a temporal logic formula,

$$p \supset \bigcirc\neg p,$$

where $\bigcirc$ is a temporal operator representing the next time. This cannot be done directly by the classical logic.

From a historical point of view (see [Rescher 71]), temporal logic was initially investigated by philosophers and logicians since Prior constructed the first *tense logic* in the 1950s. Then, since the 1970s, temporal logic has been applied to computer science, especially, artificial intelligence[1] and software engineering[2] . Temporal logic is an especially useful formal framework for *specification*, *verification*, and *synthesis* for reactive and concurrent systems. The application of temporal logic to these systems has been actively investigated by many researchers all over the world from the late 1970's to the present. The historical background of temporal logic was well surveyed by Galton [Galton 81].

[1] For example, McDermott's Temporal Logic, and Allen's Theory of Time.
[2] Pnueli initially systematized temporal logic [Pnueli 77] in software engineering.

In this thesis, we adopt temporal logic as a formal specification language for reactive and concurrent systems for the following reasons.

- The modality of temporal logic is intuitive and easy for software designers to understand.
- A temporal logic formula can be translated into an equivalent *ω-automaton* which is compatible with a state transition system and can be easily manipulated in verification and synthesis.
- Most reactive and concurrent systems can be basically modeled by state transition systems.

Temporal logic has been applied to *specification*, *verification*, and *synthesis* for reactive and concurrent systems. First, the research background on specification by temporal logic is briefly surveyed.

There are several variants of temporal logic. The variant initially presented by Pnueli is *linear time temporal logic* , which has been investigated chiefly by Pnueli, Manna, and their group [Pnueli 77, Pnueli 81, Manna 81a, Manna 81b, Pnueli 86, Manna 92] for specification and verification of concurrent programs. A second common version of temporal logic is *branching time temporal logic* [Ben-Ari 83, Emerson 85a]. Each logic assumes a different underlying nature of time, as follows [Emerson 90a].

- **Linear time temporal logic:** The course of time is linear; at each moment there is only one possible future moment.
- **Branching time temporal logic:** Time has a branching, tree-like nature; at each moment, time may split into alternate courses representing different possible futures.

Linear time temporal logic is suitable for specification for program synthesis, while branching time temporal logic is suited to describe queries for verification. Besides these two popular version, there are several versions of temporal logic for concurrent programs (e.g., Partial Order Temporal Logic [Pinter 84], Interval Temporal Logic (ITL) [Moszkowski 86]).

These logics are sufficient to consider the essential features of reactive and concurrent systems, but not expressive enough to describe an entire system specification. There are two approaches to make temporal logic suitable for practical specifications; extension of temporal logic and combination of temporal logic with other formalisms (called *dual-language approach* in [Felder 94]).

With regard to the first approach, temporal logic can be extended by introducing the following features for practical specifications.

- regular expression [Wolper 83b]
- more than operator [Yoshimura 93]
- unbounded message buffers [Sistla 84, Koymans 87]
- modularity, compositionality and abstractness [Barringer 84, Josko 87, Yonezaki 91]
- nonmonotonicity for avoiding frame problem [Saeki 87]
- real-time [Alur 89, Ostroff 90]

Furthermore, several executable specification languages based on temporal logic have been proposed, influenced by logic programming (e.g. Tempula [Moszkowski 86]), in which operational semantics can be given to extended temporal logic formulas. These executable specification languages are useful for prototyping of reactive and concurrent systems [Hale 87]. However, in spite of these extensions, it is still difficult to describe an entire practical system by temporal logic. Moreover, these extensions often make automatic verification and synthesis difficult, and increase computing costs.

With regard to the second approach, temporal logic is combined with another formalism as a specification language. Since temporal logic is declarative, a combination of temporal logic and other formal language having operational semantics, like transition system and Petri net is effective. A combination of temporal logic and Petri net has been investigated recently by our group and others. These works will be surveyed in Chapter 4. The second approach appears promising and realistic for practical specification.

2.2 Verification for Reactive and Concurrent Systems

The background on the research on verification by temporal logic is briefly surveyed. There are two approaches in verification for reactive and concurrent systems; logical reasoning and model-checking.

- **Logical Reasoning Approach:**

  Both specification and implementation (program) are specified by temporal logic formulas. If the specification is represented by a formula f_s and the implementation is represented by a formula f_i, then the implementation is correct iff $f_i \supset f_s$ is valid. Manna and Pnueli [Manna 81b, Pnueli 81] presented methods for proving $f_i \supset f_s$ inductively using axioms and inference rules. Furthermore, Manna and Wolper [Manna 84] presented an automatic verification method based on refutation, where it is shown that $\neg(f_i \supset f_s)$ is not satisfiable using tableaux construction.

- **Model Checking Approach:**

  In a model-checking approach, the specification is represented by a temporal logic formula f, and the implementation is represented as a model M of temporal logic. The implementation is correct iff M is a model of f (i.e., $M \models f$). Transition systems are usually used to represent models (i.e, implementations). Clarke, Emerson, and Sistla [Clarke 86] initially proposed an automatic verification method of finite-state concurrent systems based on model-checking. They dealt with a branching time temporal logic called CTL (Computation Tree Logic).

Since the model-checking approach can provide a practical and widely applicable verification method as compared with the logical reasoning approach, much work based on model-checking has been done over the last decades. The early researches in this area are well surveyed in [Clarke 87]. The latest trends can be caught by watching *the Annual Conference on Computer-Aided Verification (CAV).* These works can be classified into two types;

- Extension of expressive ability of temporal logics and models (e.g., Fairness [Emerson 85b])
- Local model-checking for Petri nets [Bradfield 92]
- Proposal of efficient model-checking algorithms

Since model-checking can be regarded as a type of state space analysis method, the *state explosion problem* is usually the limiting factor in applying these algorithms to realistic systems. Therefore, an efficient model-checking algorithm is required. These efficient algorithms can be classified into the following three approaches.

- Symbolic Model Checking [Burch 90, McMillan 93]
- Partial Order Approach [Valmari 90, Godefroid 91a, Godefroid 96]
- Compositional Approach [Clarke 89]

This thesis will focus on the compositional approach, and propose a new compositional verification method.

2.3 Synthesis for Reactive and Concurrent Systems

The first attempts to synthesize reactive and concurrent systems from temporal logic specifications were developed in [Manna 84] and [Emerson 82]. Propositional versions of linear time temporal logic and branching time temporal logic are used as the specification language in [Manna 84] and [Emerson 82], respectively. Both synthesis methods are based on tableaux construction representing models of the given temporal logic formula. The target finite-state program is generated from this tableaux.

These pioneering synthesis method can be applied to the synthesis of *closed* reactive systems. The closed reactive system means a system in which all actions are observable and controllable. In contrast, *open* reactive systems has unobservable and uncontrollable actions. For example, a plant control system consists of a controller and controlled objects in which some actions and states are unobservable and uncontrollable from the controller. Pnueli and Rosner proposed a synthesis method for these open reactive systems [Pnueli 89a, Pnueli 89b, Pnueli 90].

These works made worthy contributions from the theoretical view point. However, from the practical point of view, they were suggestive but hard to apply to actual systems. The main reason is that it is too difficult and too expensive to describe a whole specification by temporal logic and then synthesize a whole program from it. Therefore, a more practical approach is proposed, "program adjustment", which modifies (adjusts) a given target program to satisfy a temporal logic specification instead of generating a whole program from it.

2.4 Design Methodology for Reactive and Concurrent Systems

Software design methodology plays important part so that formal methods are utilized during the software development process. As a software design methodology for reactive and concurrent systems, several methods have been proposed; Real-Time Structured Analysis and Design (RTSAD), Object-Oriented Analysis and Design (OOAD), and Design Approach for Real-Time Systems (DARTS).

- **Real-Time Structured Analysis and Design (RTSAD)**

  Real-Time Structured Analysis and Design (*RTSAD*) is an extension of Structured Analysis and Design for real-time systems. There are two popular variations which have been developed by Ward [Ward 85, Ward 86] and Hatley [Hatley 87], respectively. In RTSAD, functional requirements for the target system are hierarchically decomposed into several functions, which are described by **data/control flow diagrams**. In addition to the data/control flow diagrams, behavioral requirements are represented by **state transition diagrams**. Since the data flow diagram is familiar to designers and easy to understand, RTSAD has been used on a wide variety of projects and there is much experience in applying RTSAD. However, RTSAD is weak in its provision of task structuring guidelines which address how to structure the system into concurrent tasks.

- **DARTS**

  In the reactive and concurrent system design, task structuring is given considerable weight. *DARTS* (Design Approach for Real-Time Systems), which was proposed by Gomaa [Gomaa 93], is a design method based on RTSAD and emphasizes the decomposition of a real-time system into concurrent tasks. DARTS provides a set of **task structuring criteria** for structuring a real-time system into concurrent tasks, as well as guidelines for for defining the interfaces between tasks.

- **Object-Oriented Analysis and Design (OOAD)**

  Object-Oriented Analysis and Design (*OOAD*) [Booch 94, Rumbaugh 91] is a design method based on object-oriented paradigm. In OOAD, classes and objects are first identified by analyzing the problem domain, then **object diagrams** and **class diagrams** are developed to describe the relationships between classes and objects. After defining the object structure, behavior for each object is described by the **state transition diagram**. Finally, objects are classified into tasks (active objects) and packages (passive objects) of the concurrent programming language.

Generally speaking, each design method consists of several *charts* which represent functional, behavioral, and module structures of systems and *guidelines (criteria)* for deriving these charts. Data/control flow diagrams, state transition diagrams, and class/object diagrams are used as the design charts in the above methods.

Although a Petri net can be a promising design chart for reactive and concurrent systems, there are few design methods which use Petri nets as the design chart and give guidelines for manipulating them. Reisig [Reisig 92] proposed a design method based on Petri nets. It is very suggestive but not sufficiently mature. This thesis considers a Petri-net-oriented design method, which is called *Net-Oriented Design method* (NOD) [Honiden 92].

2.5 Programming Environments for Reactive and Concurrent Systems

A lot of tools have been proposed which support formal methods for reactive and concurrent systems. Recently, these tools have become sophisticated and are of practical use[3] . The more well-known tools include the following.

[3] Some of them can be easily obtained by INTERNET.

- Tools based on state transition systems (state machine, automata):

  STATEMATE[4] [Harel 90] and *SPIN* [Holzmann 91],

- Tools based on process algebra[5] :

  Concurrency Workbench (*CWB*) [Cleaveland 93] and *AUTO/AUTOGRAPH* [Boudol 89],

- Tools based on Petri nets:

  DESIGN/CPN [Jensen 92] and *Cabernet* [Ghezzi 93].

However, none of them is a comprehensive programming environment covering the overall development process which includes verification, synthesis, and a design methodology based on *Petri nets* and *temporal logic*.

3 Synopsis

The organization of this thesis is as follows (Fig. 2).

Chapter 2 starts by providing definitions, models, and logics for reactive and concurrent systems as a preliminary section.

Chapter 3 considers a software development process for reactive and concurrent systems. In this chapter, a definition and the characteristics of reactive and concurrent systems and a development process for them are discussed. Finally, a software development process using Petri nets and temporal logic is conceptually proposed. According to this conceptual development process, detailed techniques are described in the following chapters.

In **Chapter 4**, a fusion of Petri nets and temporal logic is proposed as a specification language for reactive and concurrent systems. Its expressive power and theoretical results are also considered. In succession, verification and synthesis methods are shown as an application of specification by Petri nets and temporal logic. Examples are provided to show the effectiveness of the verification and synthesis. The remarkable point is that proposed methods are for unbounded Petri nets, while former verification and synthesis methods were mainly for bounded (safe) Petri nets.

Chapter 5 considers an efficient and practical verification method, temporal logic model-checking, for finite-state (i.e. bounded) systems. The model-checking method is actually useful to verify reactive and concurrent systems. However, as the scale of the programs increases, the computation costs for verification increase exponentially due to the state explosion. To ease the state explosion, a compositional approach to model-checking seems promising and is adopted here. The point of compositional verification is to reduce (localize, minimize) the target program, leaving only essential information for each verification query. The reduction of the target program is formalized by process equivalence theory. This chapter introduces a new process equivalence relation ($\pi\tau\omega$-bisimulation equivalence) for compositional verification. This new relation is required because conventional bisimulation equivalence, which was used in other compositional verification, cannot deal with "divergence" explicitly. An explanation of why conventional bisimulation equivalence does not work well with divergence is given, and then $\pi\tau\omega$-bisimulation is defined. After this, Process Query Language (PQL) is proposed. PQL is a modal logic which is union of temporal logic and process logic, and the semantics of which is based on $\pi\tau\omega$-bisimulation equivalence. Then, this chapter proposes the compositional verification method using PQL with consideration of the divergence. Its effectiveness is demonstrated by means of some experimental results.

In **Chapter 6**, program synthesis using temporal logic is discussed. From the standpoint that the conventional program generation approach from temporal logic specification, which was initially proposed by Manna and Wolper [Manna 84], is impractical, this chapter proposes a new approach, "program adjustment". In program adjustment, a target program made by programmers, which may be functionally correct but may be imperfect in its timing, is automatically adjusted (tuned up) to satisfy given temporal logic constraints. To put it concretely, program adjustment is realized by adding an arbiter process which is synchronized with and restricts the behavior of the target program. It is more feasible for ordinary programmers to adopt the program adjustment approach compared to conventional program generation approach for the following reasons.

[4] STATEMATE is a trademark of i-Logix

[5] These tools are also applicable for state transition systems which are equivalent to the process algebra.

- It is not very difficult for ordinary programmers to produce a target program, which satisfies at least the functional requirements. A more difficult task is to design and debug the timing of such programs.
- It is easy for ordinary programmers to specify timing constraints, such as deadlock-free and starvation-free constraints, as compared with implementing them.
- Computation cost of program adjustment is generally smaller than program generation.

Furthermore, when a target program becomes large, the arbiter synthesis may cause a computing cost explosion. Therefore, we propose compositional adjustment.

Chapter 7 considers Petri nets as a programming language instead of a specification language. First, it is discussed what properties are required for a programming language for reactive and concurrent systems, and what extensions are required to use Petri nets as a programming language. Then, a MENDEL net is introduced, which is a high-level Petri net for reactive and concurrent systems. A MENDEL net can be used not only for specification but also for programming (detail prototyping) owing to several extensions, an I/O interface with the environment, concurrent tasks, and a mechanism for their scheduling.

Chapter 8 proposes a software design methodology based on MENDEL nets and temporal logic, which is called *Net-Oriented Design method* (NOD). In this method, a *causality matrix* is introduced at an earlier design phase when the system structure is obscure and it is difficult to write MENDEL nets directly. According to the design method utilizing causality matrices, a designer can construct MENDEL nets systematically from an ambiguous requirement, then verify and adjust the MENDEL nets by methods based on temporal logic mentioned in earlier chapters.

Chapter 9 shows an overview of a programming environment for reactive and concurrent systems, called *MENDELS ZONE*. MENDELS ZONE provides analysis tools based on formal methods (e.g., verification and adjustment tools) and a design support tool based on the informal design method (e.g., causality matrix editor) in addition to the usual programming tools such as graphical MENDEL net editor, simulator, and compiler. MENDELS ZONE has been implemented on the parallel machine, Multi-PSI.

Finally, **Chapter 10** concludes this thesis, summarizing the research contributions, and presenting intentions for further work.

Related our publications are summarized as follows.

Chapter 4: [Uchihira 90b]

Chapter 5: [Uchihira 92a]

Chapter 6: [Uchihira 92c, Uchihira 95a]

Chapter 7: [Uchihira 96b]

Chapter 8: [Uchihira 92b]

Chapter 9: [Uchihira 87, Uchihira 88, Uchihira 90a, Uchihira 92b, Uchihira 95a]

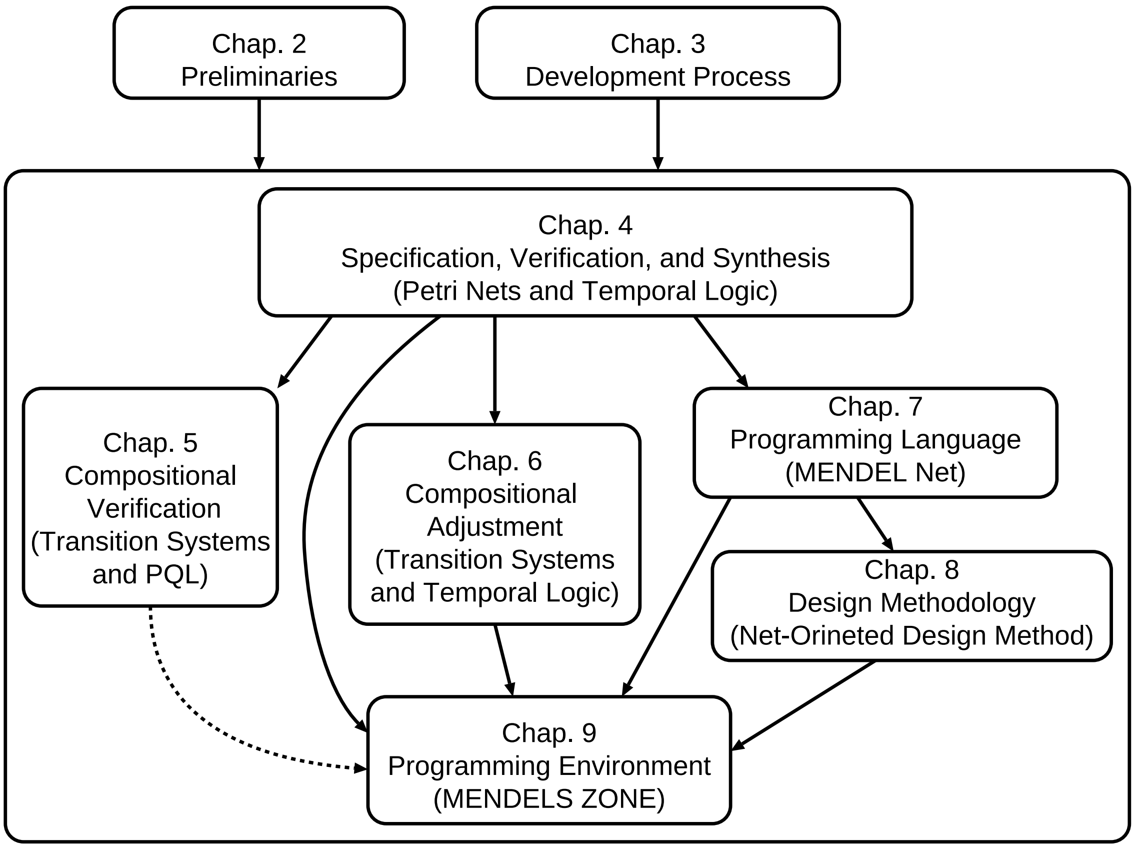


Figure 2. Organization of The Thesis

Chapter 2

Preliminaries: Models and Logics for Reactive and Concurrent Systems

This chapter introduces several notations and definitions of models and logics for reactive and concurrent systems.

1 Models for Reactive and Concurrent Systems

Many models have been proposed for reactive and concurrent systems. Most of them are based on state transition systems which may be extended about *concurrency* and *liveness*. Among them, this section shows Transition System, Büchi Sequential Automaton, Finite State Process, and Petri Net.

1.1 Sequence

To begin with, we introduce several notations about sequences. In the context of this thesis, a sequence is used to represent the time series of actions which a reactive and concurrent system takes, and states which it stays in.

Let X be a set. The set of all finite sequences over X, including the empty sequence ε, is denoted by X^*. If there is no empty sequence ε, the set is denoted by X^+. The set of all infinite sequences over X is denoted by X^ω; ω means "infinitely many". X^∞ is defined by $X^\infty = X^* \cup X^\omega$. $\overline{X}$ means a complementary set of X. $X_1 \backslash X_2 = X_1 \cap \overline{X_2}$

For a sequence $\theta \in X^\infty$, $\theta[i]$ means the i-th element in θ; $\theta(k)$ means the prefix subsequence $\theta[1]\theta[2]...\theta[k]$ of θ, and $\mid \theta \mid$ the length of θ.

For $A \subset X$ and $L \subset X^\infty$, L/A is defined as $L/A \overset{def}{=} \{\theta' \mid \exists \theta \in L.\forall i.(\theta'[i] = \theta[i] \text{ if } \theta[i] \in A, \text{ otherwise } \theta'[i] = \varepsilon)\}$. "/" is a label restriction operator. Intuitively, L/A consists of sequences of L in which all elements except A are deleted.

1.2 Transition System

Definition 1 (Transition System)
$TS = (S, P, A, \pi, \delta, s_0)$
S : *A set of states*
P : *A set of state attributes*
A : *A set of actions*
$Act = A \cup \{\tau\}$
τ : *an unobservable internal transition*
$\pi : S \to 2^P$ *A boolean function*
$\delta : S \times Act \to 2^S$ *A nondeterministic transition function*
s_0 : *An initial state*

For $TS = (S, P, A, \pi, \delta, s_0)$ and $s, s' \in S$ and $a \in A$, $p \in \pi(s)$ means that a state attribute p is true in s, and $s' \in \delta(s, a)$ means that the system which stays in s can move to the state s' after an action a.

According to Milner's notation [Milner 89], $\delta(s,a) \ni s'$ is expressed as $s \xrightarrow{a} s'$, $s(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* s'$ is expressed as $s \overset{a}{\Rightarrow} s'$. Also, $\hat{a}$ expresses a when $a \neq \tau$, and $\hat{a}$ expresses ε when $a = \tau$. Here, ε means an empty string, that is, $s \overset{\hat{\tau}}{\Rightarrow} s' = s \overset{\varepsilon}{\Rightarrow} s' = s(\xrightarrow{\tau})^* s'$. We show a simple example of a transition system in Figure 3.

Example 1 (Transition System) $T = (\{s_0, s_1, s_2, s_3\}, \{p_1, p_2\}, \{a, b\}, \pi, \delta, s_0)$ *where*
$s_0 \xrightarrow{a} s_1, s_0 \xrightarrow{b} s_2, s_1 \xrightarrow{b} s_3, s_2 \xrightarrow{a} s_3, s_3 \xrightarrow{\tau} s_0$, *and* $\pi(s_0) = \{p_1, p_2\}, \pi(s_1) = \{p_1\}, \pi(s_2) = \{p_2\}, \pi(s_3) = \emptyset$.

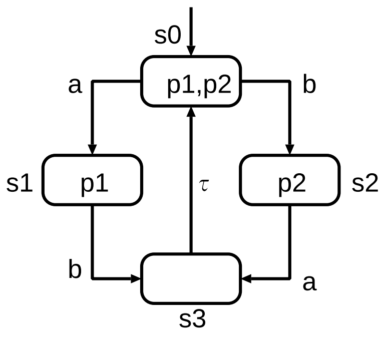


Figure 3. Example of a transition system

Definition 2 (Finite Branching Condition)
A finite branching condition in $TS = (S, P, A, \pi, \delta, s_0)$ *is:*

$$\delta(s,a) \text{ is a finite set for } \forall s \in S \text{ and } \forall a \in A.$$

This finite branching condition is necessary for the fully automatic verification and synthesis for transition systems. When S, P, and A are finite, the finite branching condition is satisfied. We assume S, P, and A in TS are finite whenever we do not mention it explicitly.

1.3 Büchi Sequential Automata

A transition system can model all possible (safe) behaviors of systems, but cannot model desirable behaviors. A finite automaton can express desirable (acceptable) behaviors explicitly using terminal states.

Definition 3 (Finite Automaton) *A finite automaton is a tuple* $A = (\Sigma, S, \rho, s_0, F)$, *where*

- Σ *is an alphabet,*
- S *is a set of states,*
- $\rho : S \times \Sigma \rightarrow 2^s$ *is a nondeterministic transition function,*
- $s_0 \in S$ *is an initial state, and*
- $F \subset S$ *is a set of terminal states.*

A run of a finite automaton A over a finite word $\theta = t_1 t_2 ... t_n \in \Sigma^*$ is a state sequence $s_0, s_1, ..., s_n$, where $s_i \in \rho(s_{i-1}, t_i)$ for all $i \geq 1$. A run $s_0, s_1, ..., s_n$ is accepting if $s_n \in F$. A finite word θ is accepted by A if there is an accepting run of A over θ. The set of all words, accepted by A, is denoted $L(A)$.

A finite automaton treats only finite sequences (words). Since reactive and concurrent systems often takes infinite ongoing computation, a finite automaton on infinite sequences is necessary to model them.

Definition 4 (Büchi Sequential Automaton) *Büchi sequential automaton is a tuple* $A = (\Sigma, S, \rho, s_0, F)$, *where*

- *Σ is an alphabet,*
- *S is a set of states,*
- *$\rho : S \times \Sigma \to 2^s$ is a nondeterministic transition function,*
- *$s_0 \in S$ is an initial state, and*
- *$F \subset S$ is a set of designated states.*

A run of a Büchi sequential automaton A over an infinite word $\theta = t_1t_2... \in \Sigma^\omega$ is a sequence $s_0, s_1, ...,$ where $s_i \in \rho(s_{i-1}t_i)$ for all $i \geq 1$. A run $s_0, s_1, ...$ is accepting if for some $s \in F$ there are infinitely many i's such that $s_i = s$. An infinite word θ is accepted by A if there is an accepting run of A over θ. The set of all words, accepted by A, is denoted $L(A)$.

1.4 Finite State Process

A finite state process [Kanellakis 90] is defined as a general model which includes both a transition system and an automaton. Therefore a finite state process can specify a transition system with *liveness conditions* as its acceptance condition.

Definition 5 (Finite State Process) *A Finite State Process (FSP) is a seventuple $P = (S, A, L, \delta, \lambda, s_0, F)$, where:*

- *S is a finite set of states,*
- *A is a finite set of actions,*
- *L is a finite set of synchronization labels,*
- *$\delta : S \times A \to S \cup \{\perp\}$ is a* **deterministic** *transition function ($\delta(s, t) = \perp$ means the action $t \in A$ is disabled in the state $s \in S$),*
- *$\lambda : A \to (L \cup \{\tau\})$ is a labeling function, (τ is an invisible internal action),*
- *$s_0 \in S$ is an initial state, and*
- *$F \subset S$ is a set of designated states.*

Example 2 (Finite State Process) *$P = (\{s_0, s_1, s_2, s_3\}, \{t_1, t_2, t_3\}, \{a, b\}, \delta, \lambda, s_0, \{s_3\})$ is a finite state process where $\delta(s_0, t_1) = s_1, \delta(s_0, t_2) = s_2, \delta(s_1, t_2) = s_3, \delta(s_2, t_1) = s_3, \delta(s_3, t_3) = s_0, \lambda(t_1) = a, \lambda(t_2) = b, \lambda(t_3) = \tau$. (Fig.4)*

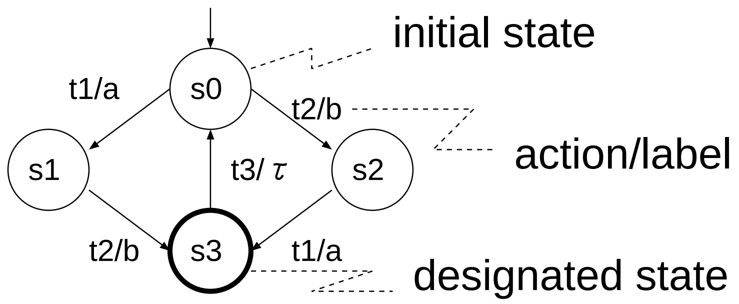


Figure 4. Finite State Process

Let $P = (S, A, L, \delta, \lambda, s_0, F)$ be an FSP. A transition function can be extended such that $\delta : S \times A^* \to S \cup \{\perp\}$, i.e., $\delta(s, \theta a) \overset{def}{=} \delta(\delta(s, \theta), a)$. Note, $\delta(s, \varepsilon) = s$. Since a transition function is deterministic, a current state can be uniquely determined from an initial state and an action sequence. We call an action sequence a *behavior*. Similarly, we can extend a labeling function such that $\lambda : A^* \to (L \cup \{\tau\})^*$, i.e., $\lambda(\theta) = \lambda(\theta[1])\lambda(\theta[2])...\lambda(\theta[\mid \theta \mid])$. In addition, $\hat{\lambda}(\theta)$ is defined as the sequence gained by deleting all occurrences of τ from $\lambda(\theta)$. The set of reachable states from a state s in P is defined as $R_P(s) \overset{def}{=} \{s' \in S \mid \exists\theta \in A^*.s' = \delta(s,\theta)\}$ and $R_P^+(s) \overset{def}{=} \{s' \in S \mid \exists\theta \in A^+.s' = \delta(s,\theta)\}$. Also, the set of all possible action sequences of P is defined as $L(P) \overset{def}{=} \{\theta \in A^* \mid \delta(s_0,\theta) \neq \bot\}$, and the set of all possible label sequences is defined as $L_\lambda(P) \overset{def}{=} \{\hat{\lambda}(\theta) \in L^* \mid \theta \in L(P)\}$. Since interest is in the infinite behavior of an FSP, we introduce a set of infinite action sequences $L_\omega(P) \subset A^\omega$ and $L_{\Delta\omega}(P) \subset (A^\omega \cup A^*\{\Delta\}^\omega)$ where Δ means *deadlock*:

$$L_\omega(P) \overset{def}{=} \{\theta \in A^\omega \mid 1 \leq \forall k.\delta(s_0,\theta(k)) \neq \bot\}$$

$$L_{\Delta\omega} \overset{def}{=} \left\{ \begin{array}{l} \{\theta \in A^\omega \mid 1 \leq \forall k.\delta(s_0,\theta(k)) \neq \bot\} \cup \\ \{\theta \in A^*\{\Delta\}^\omega \mid \exists k.(\left\{ \begin{array}{l} 1 \leq \forall i \leq k.\delta(s_0,\theta(i)) \neq \bot \text{ and} \\ \forall a \in A.\delta(\delta(s_0,\theta(k)),a) = \bot \text{ and} \\ \theta[j] = \Delta \text{ for } \forall j > k \end{array} \right\})\} \end{array} \right.$$

$L_{\Delta\omega}(P)$ is an extension of $L(P)$ into a set of infinite action sequences where if $\theta \in L(P)$ is a deadlock sequence (i.e., an inevitably finite sequence), then θ is represented as $\theta\Delta^\omega \in L_{\Delta\omega}(P)$.

$L_{\Delta\omega}^{fair}(P) \subset L_{\Delta\omega}(P)$ is defined as $L_{\Delta\omega}^{fair}(P) \overset{def}{=} \{\theta \mid \theta \in L_{\Delta\omega}(P)$ under the fairness condition$\}$ where the *fairness condition* means whenever a behavior θ infinitely often passes through some state s, every action a enabled at s must appear infinitely often on θ (i.e., if $s = \delta(s_0,\theta(i))$ for infinitely many i and $\delta(s,a) \neq \bot$, then $s = \delta(s_0,\theta(j))$ and $\theta[j+1] = a$ for infinitely many j).

An FSP is a transition system with liveness conditions. In an FSP, liveness conditions are represented by designated nodes that indicate *satisfiable behavior* of an FSP as follows.

Definition 6 (Satisfiable Behavior) *Let $P = (S,A,L,\delta,\lambda,s_0,F)$ be an FSP. $\theta \in A^\omega$ is a satisfiable behavior, if $\delta(s_0,\theta(k)) \in F$ for infinitely many $k \geq 1$. $L_b(P) \subset A^\omega$ is defined as a set of all satisfiable behaviors on P.*

Note that a satisfiable behavior corresponds to an accepting run of Büchi automaton.

Definition 7 (Completeness of FSP)
Let $P = (S,A,L,\delta,\lambda,s_0,F)$ be an FSP. P is **complete** *if $\forall s \in R_P(s_0).\exists s' \in R_P^+(s)$ and $s' \in F$.* □

A state $s \in R_P(s_0)$, having no path to designated nodes from s, is called an *unsatisfiable state*. If P is complete, P has no unsatisfiable states. A behavior reaching an unsatisfiable state is called an *inevitably unsatisfiable behavior*.

Lemma 1 *If an FSP P is complete, then $L_{\Delta\omega}^{fair}(P) \subset L_{sat}(P)$.* □

This lemma means that if P is complete, then a random transition over P leads to a satisfiable behavior. Consequently, if P is complete, P is deadlock-free.

1.5 Transition System with Concurrency

Transition systems (finite state processes, automata) have no ability to express concurrency explicitly by themselves. Reactive and concurrent systems are constructed from some number of processes. Each local process can be modeled as a transition system. When processes are modeled by transition systems $TS_1, TS_2, ..., TS_n$, concurrency among processes can be expressed as process composition $TS_1 \mid TS_2 \mid ... \mid TS_n$ using composition operators '|'. $TS_1 \mid TS_2$ means a concurrent system in which TS_1 and TS_2 run concurrently and communicate with each other. It is called *communicating transition systems*.

Furthermore, transition systems are also used to define semantics of process composition. Global behaviors of communicating transition systems can be expressed by one global transition system based on *interleaving semantics*. In the interleaving semantics, concurrent structures of systems is expanded into a set of nondeterministic global behaviors.

Process composition and interleaving semantics will be explained in detail in Chapter 5 for transition systems and in Chapter 6 for finite state processes.

I/O automata and *Statechart* can be classified into a kind of communicating transition systems. I/O automata is communicating transition systems featuring reactive properties. In I/O automata [Lynch 86, Lynch 88], each process is modeled by an automaton with input and output actions. Due to reactive properties, input actions cannot be controlled by the automaton, while output actions can be. Statecharts [Harel 87a] are extended transition systems in order to design large and complex reactive and concurrent systems. Statecharts have several extended features including hierarchy, concurrency, and their visual formalisms. For example, a system which consists of three processes TS_1, TS_2, TS_3 can be expressed in Fig. 5. Since Statechart can express concurrency graphically, it is often used in design methodologies and CASE tools.

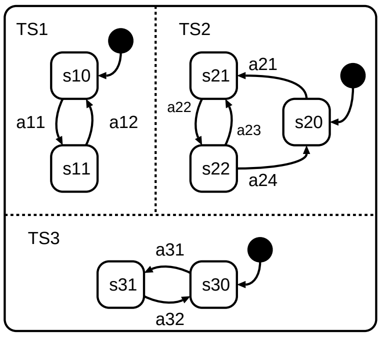


Figure 5. Concurrency in Statechart

1.6 Petri Net

A *Petri net* is another approach to express concurrency explicitly in addition to state transition models.

Definition 8 (Petri Net) *A Petri net is a 4-tuple $N = (P, T, w, m_0)$ where:*

- $P = \{p_1, p_2, ..., p_n\}$ *is a finite set of places,*
- $T = \{t_1, t_2, ...t_m\}$ *is a finite set of transitions,*
- $w : (P \times T) \cup (T \times P) \to \{0, 1, 2, 3, ...\}$ *is a weight function, $w(p, t)$ is the weight of the arc from p to t and $w(t, p)$ is the weight of the arc from t to p, especially $w(p, t) = 0$ ($w(t, p) = 0$) means there exists no arc between p and t,*
- $m_0 : P \to \{0, 1, 2, ...\}$ *is the initial marking.*
- $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$

A marking in a Petri net is changed according to the following firing rules:

- *A transition is said to be enabled, if each input place p of t is marked with at least $w(p, t)$ tokens.*
- *Only one of the enabled transitions can fire at a time.*
- *Firing of an enabled transition t removes $w(p, t)$ tokens from each input place p of t, and adds $w(t, p)$ tokens to each output place p of t.*

Let t be a transition and $P = \{p_1, p_2, .., p_n\}$ be a set of places. An n-dimensional differential vector is defined as $\Delta(t) = (w(t, p_1) - w(p_1, t), ..., w(t, p_n) - w(p_n, t))$. Furthermore, $\Delta(t_1 t_2 ... t_m) = \Delta(t_1) + \Delta(t_2) + ... + \Delta(t_m)$ for a transition sequence $t_1 t_2 ... t_m$.

When $t \in T$ is enabled at a marking m, we denote $m[t>$. After firing t, if m' is a new marking, we denote $m[t> m'$. In the case that $m_1[t_1 > m_2, m_2[t_2 > m_3, ..., m_{k-1}[t_{k-1} > m_k$ for a transition sequence $\theta = t_1 ... t_{k-1}$, we denote $m_1[\theta > m_k$. A set of reachable markings $R(N)$ of Petri net N is defined as $R(N) = \{m \mid \exists\theta \in T^*.m_0[\theta > m\}$.

Definition 9 (Labeled Petri net) *A labeled Petri net is a 2-tuple, $LN = (N, \lambda)$, where $N = (P, T, w, m_0)$ is a Petri net and $\lambda : T \rightarrow \Sigma$(alphabet) $\cup \{\varepsilon$(empty sequence)$\}$ is a labeling function.*

The labeling function $\lambda : T \rightarrow \Sigma \cup \{\varepsilon\}$ is extended to $\lambda : T^\infty \rightarrow \Sigma^\infty$ by $\lambda(\theta)[i] = \lambda(\theta[i])$ for all $\theta \in T$ and $1 \leq i \leq\mid \theta \mid$. A sequence of transitions ($\theta \in T^*$) is called a *legal firing sequence* on the Petri net N if the firing sequence θ is allowed by the firing rules; an infinite sequence of the transitions ($\theta \in T^\omega$) is a legal firing sequence if every prefix is a legal firing sequence. The set of all finite (infinite) legal firing sequences of N is denoted by $F(N)$ ($F_\omega(N)$).

Definition 10 (Petri net language) *$L(N, \lambda)$ is a finite Petri net language generated from a labeled Petri net (N, λ) if $L(N, \lambda) = \{\lambda(\theta) \in \Sigma^* \mid \theta \in F(N)\}$. Similarly, $L_\omega(N, \lambda) = \{\lambda(\theta) \in \Sigma^\omega \mid \theta \in F_\omega(N)\}$ is an infinite Petri net language, and $L_{\Delta\omega}(N, \lambda) = \{\lambda(\theta) \in \Sigma^\omega \mid \theta \in F_\omega(N)\} \cup \{\lambda(\theta) \in \Sigma^* \mid \theta \in F(N)$ and $\forall t \in T.\theta t \notin F(N)\}$ is a Δ-infinite Petri net language. Here, "Δ-infinite" means "infinite including Δ^ω".*

Remark: $L_{\Delta\omega}(N, \lambda)$ may include finite words such as $s\Delta^\omega (s \in \Sigma^*)$, while $L_\omega(LN)$ includes only infinite words.

We introduce a label manipulation operator. When $L \subset \Sigma$, we define a label restriction operator "/" for a labeling function as $\lambda/L : T \rightarrow L \cup \{\varepsilon\}$ such that $\lambda/L(t) = \lambda(t)$ if $\lambda(t) \in L$ and $\lambda/L(t) = \varepsilon$ if $\lambda(t) \notin L$. This means $h(t) \in L$ is visible and $h(t) \in \Sigma \backslash L$ is invisible. An identity function $e : T \rightarrow T$ is defined as $e(t) = t$ for all $t \in T$. We use an abbreviation $\theta/L \overset{def}{=} e/L(\theta)$.

1.7 Models for Real-Time Systems

Several models for real-time systems are proposed as extension of transition systems, automata, and Petri nets, which include timed automata (timed graphs) [Alur 90, Alur 91], timed transition systems [Henzinger 91], *generalized stochastic Petri nets* [Marsan 86, Marsan 95], time Petri nets [Berthomieu 91], and time basic nets [Bellettini 93]. In this thesis, we do not mention them in detail.

1.8 Comparison of Models

Table 1 shows comparison of models which we mentioned, with regard to concurrency and liveness.

Table 1. Comparison of Models

Models	*Concurrency*	*Liveness*
Transition System		
Transition System with composition operators	○	
Büchi Sequential Automaton		○
Finite State Process with composition operators	○	○
Statechart	○	
Petri Net	○	

2 Logics for Reactive and Concurrent Systems

This section introduces several temporal and modal logics: Linear Time Temporal Logic, Branching Time Temporal Logic, Process Logic, and Propositional Modal μ-calculus. These logics can provide useful formalisms for specifying and verifying reactive and concurrent systems.

2.1 Propositional Logic

A logical language is given by an alphabet of symbols and the definition of a set of strings over language, called formulas. The simplest kind of a logical language is called propositional logic which can be given as follows.

Syntax

Propositional logic formulas are built from:

- A set of all atomic propositions: $Prop = \{p_1, p_2, p_3, ..., p_n\}$,
- Boolean connectives: $\wedge$ and $\neg$,
- Parentheses: "(" and ")".

The formation rules are:

- An atomic proposition $p \in Prop$ is a formula,
- If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$, $\neg f_1$,
- If f is a formula, so are (f).

Abbreviation

Further logical operators and constants can be introduced to abbreviate particular formulas.

Abbreviated Formula	**= Original Formula**
$f_1 \vee f_2$	$= \neg(\neg f_1 \wedge \neg f_2)$
$f_1 \supset f_2$	$= \neg f_1 \vee \neg f_2$
$true$	$= f \vee \neg f$
$false$	$= \neg true$

2.2 Linear Time Temporal Logic

Temporal logic is a special type of modal logic, where its modalities concern with time. There two possible views regarding the underlying nature of time. One is that the course of time is linear. The other is that time has branching, tree-like nature. First, we show *Propositional Linear time Temporal Logic* (*PLTL*) based on the former view. Then, Propositional Branching time Temporal Logic based on the latter view will be shown.

Syntax PLTL formulas are built from:

- A set of all atomic propositions: $Prop = \{p_1, p_2, p_3, ..., p_n\}$
- Boolean connectives: $\wedge$, $\vee$
- Temporal operators: $\bigcirc$ ("next"), U("until")

The formation rules are:

- An atomic proposition $p \in Prop$ is a formula.
- If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$, $\neg f_1$, $\bigcirc f_1$, $f_1 U f_2$.

Semantics The operators intuitively have the following meanings: $\Box f$ (read next f): f is true for the next state, $f_1 U f_2$ (read f_1 until f_2): f_1 is true until f_2 becomes true and f_2 will eventually become true. The precise semantics are given as the *Kripke structure* [Manna 84].

We use $\Diamond f$ ("eventually f") as an abbreviation for (true U f) and $\Box f$ ("always f") as an abbreviation for $\neg F \neg f$. Also, $f_1 \vee f_2$ and $f_1 \supset f_2$ represent $\neg(\neg f_1 \wedge \neg f_2)$ and $\neg f_1 \vee f_2$, respectively.

Lemma 2 *Given an PLTL formula f, one can build a Büchi sequential automaton* $A_f = (\Sigma, S, \rho, s_0, F)$, *where* $\Sigma = 2^{Prop}$, *such that* $L(A_f)$ *is exactly the set of sequences satisfying formula* f.

Proof Ref. [Wolper 83a].

Here, an accepted word $\theta = P_1 P_2 P_3 ... P_i ...$ $(P_i \subset P)$ means that all $p \in P_i$ are true and all $p \notin P_i$ are false at time i $(0 < i)$.

Definition 11 (Single Event Condition) *A single event condition is defined as follows,*

$$f_{SEC} = \Box((\bigvee_{1 \leq i \leq n} p_i) \wedge (\bigwedge_{1 \leq i < j \leq n} p_i \wedge p_j))$$

where $p_1, .., p_n$ are all atomic propositions. $\Box$

This *single event condition* provides that only just one atomic proposition is true at any moment [Manna 84]. When we build a Büchi sequential automaton $A'_f = (\Sigma, S, \rho, s_0, F)$ where f' is f with the single event condition, we can make $\Sigma = Prop$ in place of $\Sigma = 2^{Prop}$, because only one atomic proposition is true at each time.

Example 3 (A'_f) *The following $A_{f'}$ (Fig.6) is built from PLTL formula f with the single event condition (i.e. $f' = f \wedge f_{SEC}$).*

$$f = \Box(t_1 \supset \bigcirc(\neg t_1 U t_2)) \wedge \Box(t_2 \supset \bigcirc(\neg t_2 U t_1))$$

$A_{f'} = (\{t_1, t_2\}, \{s_0, s_1, s_2, s_3\}, \delta, s_0, \{s_0, s_3\})$ *where* $\delta = \{\{s_1\} = \delta(s_0, t_1), \{s_2\} = \delta(s_0, t_2), \{s_3\} = \delta(s_1, t_2)), \{s_3\} = \delta(s_2, t_1), \{s_1\} = \delta(s_3, t_1), \{s_2\} = \delta(s_3, t_2)\}$.

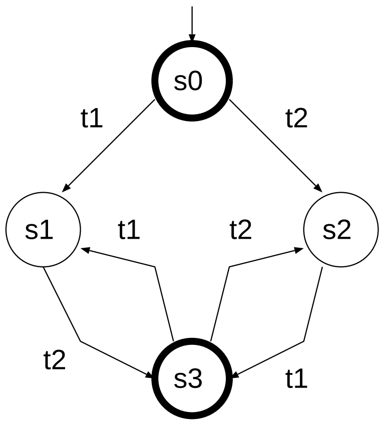


Figure 6. Büchi Automaton $A_{f'}$ built from PLTL formula f

Definition 12 *$L_s(f)$ is an infinite language generated from an PLTL formula f under the single event condition, iff $L_s(f) = L(A_{f'})$ where $f' = f \wedge f_{SEC}$, and a set of atomic propositions Prop becomes an alphabet of $A_{f'}$.*

Lemma 3 *Given an PLTL formula f, $\overline{L_s(f)} = L_s(\neg f)$, where $\overline{L_s(f)} = Prop^\omega \backslash L_s(f)$.*

Proof. The paper [Wolper 83a] proved that $\overline{L(A_{f_1})} = L(A_{f_2})$, where $f_2 = \neg f_1$ and no single event condition is assumed there. This lemma is a special case of that theorem.$\Box$

2.3 Branching Time Temporal Logic

We will consider a simple version of *propositional branching time temporal logic*: CTL (*Computation Tree Logic*).

Syntax CTL formulas are built from:

- A set of all atomic propositions: $Prop = \{p_1, p_2, p_3, ..., p_n\}$
- Boolean connectives: $\wedge$, $\vee$,
- Temporal operators: $\bigcirc$, U,
- Path operators: $\forall$, $\exists$,

The formation rules are:

State Formulas

- An atomic proposition $p \in Prop$ is a state formula.
- If f_1 and f_2 are state formulas, so are $f_1 \wedge f_2$, $\neg f_1$.
- If g is a path formula, then $\forall g$ and $\exists g$ are state formulas.

Path Formulas

- If g_1 and g_2 are state formulas, then $\bigcirc g_1$, and $g_1 U g_2$ are path formulas.

CTL Formulas

- If f is a state formula, then f is a CTL formula.

Semantics The operators intuitively have the following meanings: $\forall \bigcirc f$ = "f becomes true at the next time for all paths", $\forall f_1 U f_2$ = "f_2 becomes eventually true and f_1 is true subsequently until then for all paths",$\exists \bigcirc f$ = "f becomes true at the next time for some path", $\exists f_1 U f_2$ = "f_2 becomes eventually true and f_1 is true subsequently until then for some path".

The precise semantics of CTL formulas are defined with a tree-like structure M and a state s. A model of a CTL formula f is an infinite tree over M starting at s (we denote $(M, s) \models f$). It is known that if a given CTL formula f is satisfiable (i.e., it has at least one model), there exist a finite transition system $TS = (S, P, A, \pi, \delta, s_0)$ such that $A = \emptyset$ (i.e., without actions) and a model of f can be obtained by unwinding TS (Small Model Theorem). Therefore we can regard transition systems without actions as models of CTL formulas.

In CTL, the formulas appearing in the scope of path quantifiers are restricted to be a single temporal operators. Therefore some PLTL formulas cannot be expressed in CTL. *CTL** is extension of CTL, in which arbitrary PLTL formulas can appear in the scope of path quantifiers. An arbitrary PLTL formula f is expressed by $\forall f$ in CTL*.

2.4 Process Logic

A process logic is a kind of modal logics in which its modalities concern with actions.

Syntax *Propositional process logic (PPL)* formulas are built from:

- A set of actions: A
- Boolean connectives: $\wedge$, $\vee$
- Modal operators: $\langle a \rangle$ for $a \in A$
- A constant: *true*

The formation rules are:

- A constant *true* is a formula.
- If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$.
- If f is a formula and a is an action $(a \in A)$, the $\langle a \rangle f$ is a formula.

Semantics The operators intuitively have the following meanings: $\langle a \rangle f$ = "it is possible to execute an action a and terminate in the state satisfying f".

Formally, PPL formulas are also interpreted with a tree-like structure M and a state s. A model of a PPL formula f is an infinite tree over M starting at s (we denote $(M, s) \models f$). It is known that if a given PPL formula f is satisfiable (i.e., it has at least one model), there exist a finite transition system $TS = (S, P, A, \pi, \delta, s_0)$ such that $P = \emptyset$ (i.e., without state attributes) and a model of f can be obtained by unwinding TS (Small Model Theorem). Therefore we can regard transition systems without state attributes as models of PPL formulas.

2.5 Propositional Modal μ-calculus

μ-calculus is a variation of process logic extended with a least fixed point operator μ. Since the fixed point operator is very powerful, μ-calculus can express both temporal logic and process logic uniformly. We show the definition of propositional modal μ-calculus.

Syntax The propositional modal μ-calculus has formulas built from:

- A set of actions: A
- A set of variables: Var
- Boolean connectives: $\wedge$, $\vee$
- Modal operators: $\langle a \rangle$ for $a \in A$
- A fixed-point operator: μ

The formation rules are:

- A variable Z is a formula.
- If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$.
- If f is a formula and a is an action ($a \in A$), the $\langle a \rangle f$ is a formula.
- If f is a formula and Z is a variable ($Z \in Var$), the $\mu Z.f$ is a formula.

$\mu Z.f$ means "every Z appearing in f are replaced with f". The semantics of μ-calculus is defined in the same way as PPL.

2.6 Logics for Real-Time Systems

There are two main approaches in introducing real-time into temporal logics.

- introduction of a global clock (Explicit Clock Temporal Logic)

  ex. $a \wedge t = T \supset \Diamond(b \wedge t \leq T + 10)$, where t is a global clock.
- introduction of time bounds (Metric Temporal Logic)

  ex. $\Box_{<5} f$.

RTTL [Ostroff 90] is an extended PLTL with a global clock variable. TCTL (Timed CTL) [Alur 90] is an extension of CTL for specifying real-time systems. Models of TCTL formula correspond to timed graphs (timed automata). In this thesis, we do not mention them in detail.

2.7 Comparison of Logics

Table 2 shows comparison of logics which we mentioned, with regard to relations between logics and models.

Table 2. Comparison of Logics

Logics	*Models*	*Equivalent Models*
PLTL	infinite sequences	accepted sequences of Büchi Sequential Automaton
CTL	infinite trees	Transition System without actions
PPL	infinite trees	Transition System without state attributes
μ-calculus	infinite trees	Transition System with actions and state attributes

3 Other Related Approaches

There are a lot of other approaches for modeling and specifying reactive and concurrent systems, which include

- process algebra (CCS, CSP, ACP, π-calculus, chemical abstract machine),
- data flow model (Kahn's model),
- object-oriented model (Actor Model), and
- concurrent logic programming (Concurrent Prolog, PARLOG, GHC).

Since these models are out of scope of this thesis, we do not mention them in detail.

Chapter 3

Software Development Process for Reactive and Concurrent Systems

This chapter proposes a conceptual software development process for reactive and concurrent systems using Petri nets and temporal logic.

1 Reactive and Concurrent Systems (RCS)

1.1 What is RCS?

A reactive system was first defined by Pnueli [Pnueli 86] and Harel [Harel 87b]. They classified computerized systems into two basically different types: *Transformational Systems* and *Reactive Systems.* Transformational systems and reactive systems are defined in the book by Manna and Pnueli[Manna 92] as follows.

- **Transformational Systems (TFS):** *"A transformational program is the more conventional type of program, whose role is to produce a final result at the end of a terminating computation. "*
- **Reactive Systems:** *"A reactive program is a program whose role is to maintain an ongoing interaction with its environment rather than to compute some final value on termination."*

We remark that reactive systems include real-time systems, that is to say, a real-time system is a specific reactive system which has real-time constraints.

Examples of reactive systems are as follows:

- Embedded control systems

  ex. Home electric appliances, car electric appliances, and communication equipment (telephone and facsimile).

- Process control systems

  ex. Control systems for chemical plants, electric power plants, steel mill plant, and sewage plants.

- Computer and network operating systems

  ex. Operating systems, switching systems, and computer network control software.

- User interface management systems

  ex. Window systems for workstations and personal computers.

Most reactive systems necessarily have *concurrency* which is often represented as concurrent tasks (multitasking) since the framework of the concurrent tasks is fit to model and implement reactive systems having several kinds of interactions which should be handled concurrently. Therefore we focus on reactive and concurrent systems (RCS) in this thesis, especially embedded and process control systems.

1.2 Characteristics of RCS

We summarize the characteristics of reactive and concurrent systems as follows.

- **Reactivity** :

  RCS maintain an ongoing interaction with their environments. In the case of plant control systems, the environment is controlled objects of the plant, and the controller interacts with the controlled objects through input devices (sensors) and output devices (commands) as shown in Fig. 7.

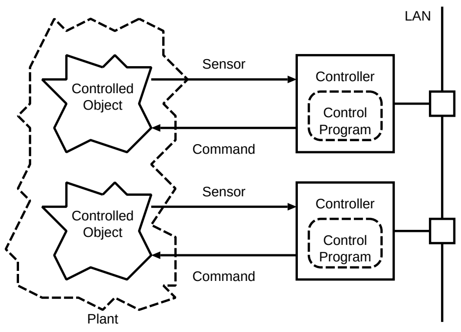


Figure 7. Plant Control System

- **Nondeterminism** :

  Behaviors of TFS are deterministic, that is, an output O can be defined as a functional relation $O = f(I)$ for a given input I in TFS. On the contrary, behaviors of RCS are nondeterministic since the environment of RCS has *uncontrollable* and *unobservable* elements. For example, timing and order of some sensory events (temperature and pressure changes) in chemical plants may be nondeterministic. These timing and order of events have an influence on the result. The nondeterministic behaviors of RCS make it difficult to test and debug the programs compared with TFS.

- **Real-time Properties** :

  Most practical RCS have real-time properties to a greater or less extent. Real-time properties include the followings.

  - **Deadline adherence:** The system should process tasks in accordance with their real-time deadlines.
  - **Periodic processing:** The tasks are activated periodically by timer events.

  The real-time systems can be classified in terms of deadline adherence.

  - **Hard real-time systems:** If the system cannot keep deadline, it may bring about a catastrophic result.
  - **Soft real-time systems:** Even if the system cannot keep deadline, it is possible to recover it by a backup method (e.g. an exception handler).

- **Concurrent Tasks** :

  The concurrent tasks are fit to model and implement RCS having several kinds of interactions which should be handled concurrently. In particular concurrent tasks are essential for real-time systems. For example, arrival of the interrupt results in a currently executing task being suspended, its contents (a program counter, stacks, etc.) being saved, and an interrupt handler to process the interrupt being invoked. After the interrupt has been serviced, the interrupted task's content is restored so that it can resume execution. It may be possible to manage these interrupts without concurrent tasks. However, a design without concurrent tasks is unnatural and makes the program difficult to debug, maintain and reuse. Moreover RCS in distributed environments (e.g. RCS which consist of several controllers connected by networks) inevitably require concurrent tasks.

In this thesis, a *task* and a *process* have the same meaning. However, we often use "task" in the context of the software design methodology and "process" in the theoretical context.

These properties (*reactivity*, *nondeterminism*, *real-time property*, *concurrent tasks*) are called "timing features of RCS" or "synchronization features of RCS", and the other features, which are in common with TFS, are called "functional features of RCS".

1.3 Software Architecture of RCS

We will consider software architectures required to construct reactive and concurrent systems.

1.3.1 Application Software and System Software

Software of a reactive and concurrent system consists of *application software* and *system software* (i.e., an operating system). The system software provides abstract manipulations of computer hardware including system resources, I/O devices, network devices, and timers (Fig. 8), and also provides concurrent tasking mechanisms including task generation, task scheduling, and task communication and synchronization. Application software is constructed on the system software by using system calls and run-time library provided by the system software.

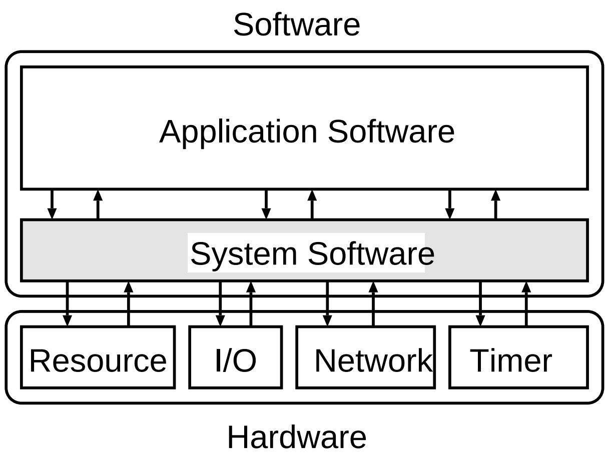


Figure 8. Application Software and System Software

In reactive and concurrent systems, *real-time operating systems* and *network operating systems* are often used as system software.

1.3.2 Concurrent Tasks

Application software of reactive and concurrent systems usually consists of several concurrent tasks. Gomaa classified them into the following tasks [Gomaa 93].

- I/O Tasks
  - Asynchronous Device I/O Tasks
  - Periodic I/O Tasks
  - Resource Monitor Task
- Internal Tasks
  - Periodic Task

- Asynchronous Task
- Control Task
- User Role Task

Concurrent tasks can be classified into I/O tasks , which correspond to I/O devices and hardware resources and manage them, and internal tasks , which supervise these I/O tasks. From another viewpoint, tasks can be classified into periodic tasks which are invoked periodically (e.g., every 100 milliseconds) and aperiodic tasks .

1.3.3 Synchronization Part and Functional Part

In most application software of RCS, a process (task) can easily be separated into two parts: a synchronization part and a functional part [Manna 84].

- **Synchronization (Timing) Part:** A part which enforces the necessary constraints on the relative timing of the execution of the different processes.
- **Functional Part:** A part which manipulates the data and performs the computation required of the program.

A functional part of RCS can be regarded as TFS. We focus on a design matter of synchronization parts; how to design and verify synchronization parts and how to adjust synchronization parts, considering functional parts.

A synchronization part design is classified into a *centralized* one and a *decentralized* one (Figure 9). A centralized synchronization part is easier to design and verify, but more difficult to implement efficiently, especially in a distributed environment. Conversely, a decentralized synchronization part is harder to design and verify, but can achieve run-time efficiency.

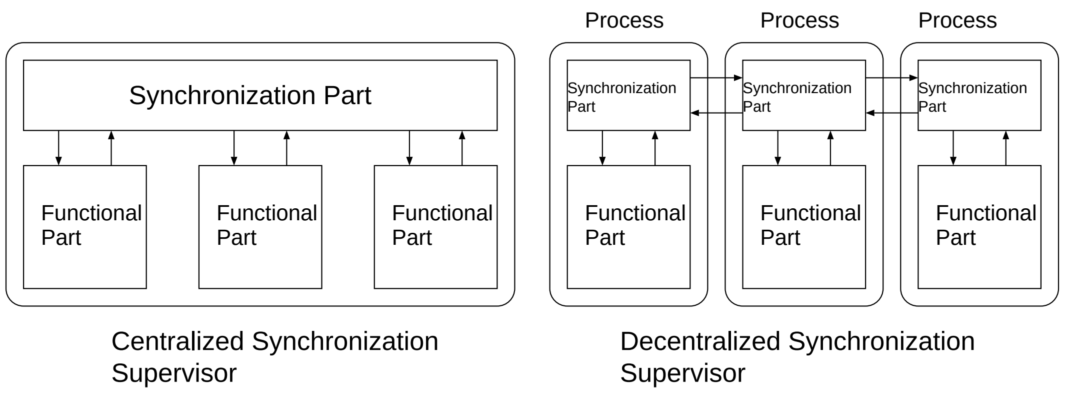


Figure 9. Synchronization Part

A hierarchical synchronization supervisor is a compromised structure between a centralized one and a decentralized one (Figure 10). It can be constructed hierarchically where each level can be constructed as a centralized one, that is, easy to design and verify. As a whole, it looks like a decentralized one.

For example, small embedded control systems such as home electric appliances adopt a structure having the hierarchical synchronization supervisor (Figure 11).

In the next section we will consider a software development process for RCS which have these characteristics and structures.

2 Software Development Process for RCS

This section takes a general view of the software development process for transformational systems (TFS) and reactive and concurrent systems (RCS), and considers a distinctive feature of RCS as compared with TFS.

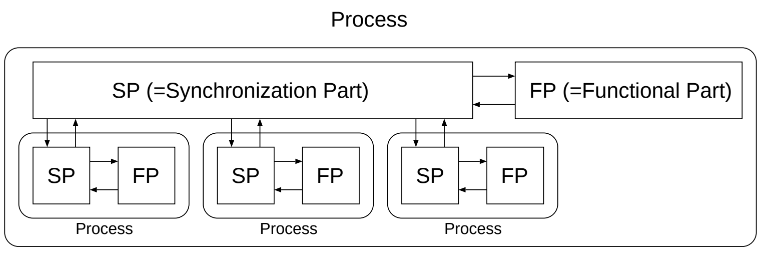


Figure 10. Hierarchical Synchronization Supervisor

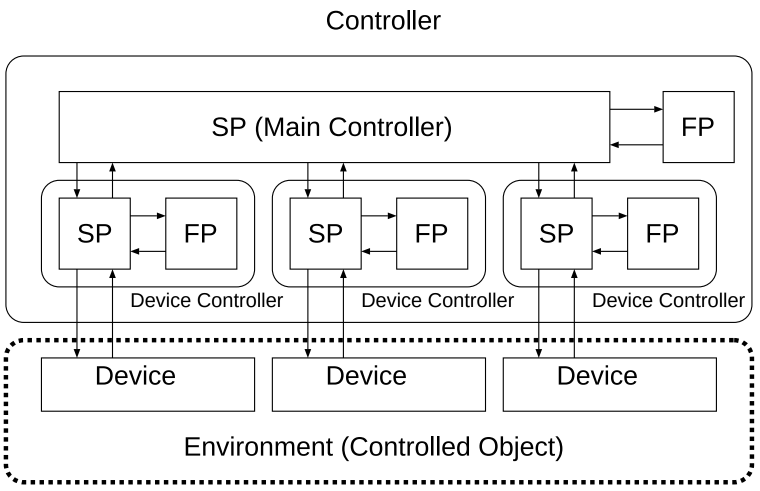


Figure 11. Hierarchical Synchronization Supervisor for Control Systems

2.1 Software Development Process for TFS

According to the software life-cycle model, the software development process can be divided into several phases. The most widely used software life-cycle model is often referred to as the “Waterfall” Model [Boehm 76]. Although the Waterfall Model has several limitations and alternative models are proposed to overcome them (Prototyping and Spiral Model), it is used here because it is sufficient to consider distinctive features of the proposed software development process for RCS.

Figure 12 shows the software development process which consists of the following phases.

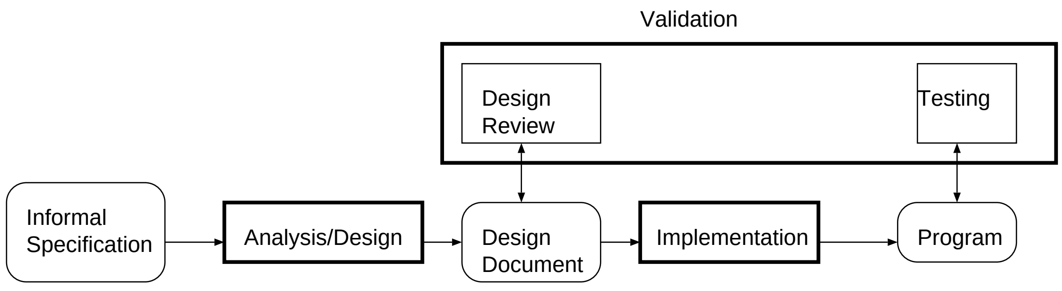


Figure 12. Software Development Process

- **Analysis/Design Phase:** This phase includes requirement analysis and architectural design. The goal of the requirement analysis is to provide a complete description of what the system's external behavior is. During the architectural design, the system's architecture is defined by design documents which describe how the system works internally. We intentionally do not divide this phase into two phases (analysis and design) clearly because analysis and design are intertwined and it is difficult to completely separate them in the actual software development.
- **Implementation Phase:** This phase includes detailed design where the algorithmic details of each component are defined and coding in a programming language.
- **Validation Phase:** This phase includes design review for design documents and testing for programs. Testing consists of unit testing, integration testing, and system testing.

2.2 Software Development Process for RCS

The software development process for RCS requires consideration of RCS characteristics(which is mentioned in the section 1.2) in addition to those of TFS.

- **Input/Output Consideration**

  Input/output considerations are important for RCS which interact with their environment through I/O devices. In the analysis/design phase, an interface between a system and its environment should be defined and designed. This includes a software design of I/O driver.

  $\rightarrow$ Reactivity

- **Concurrent Task Structuring**

  Concurrent task structuring is one of the most important issues in RCS design. Concurrent task structuring includes the designer's tradeoff between introducing tasks to simplify the design and not introducing too many tasks which increase intertask communication overhead. Concurrent task structuring criteria are needed to help the designer make the tradeoff and show him how to decompose a software system into tasks systematically.

  $\rightarrow$ Concurrent Tasks

- **Task Communication and Synchronization**

After task structuring, the task interface is defined. The task interface consists of communication and synchronization between tasks. The most popular method of task communication design is a data flow diagram (DFD) which shows communication relationships between tasks, including input data, output data, and data stores. On the other hand, task synchronization design includes deadlock prevention, mutual exclusion, and other timing issues. In the design phase, communication and synchronization are designed using message communication, event synchronization, and data stores, which are implemented later using run-time support services (mail boxes, semaphore, event flag, etc.)

→ Concurrent Tasks and Nondeterminism

- **Performance Design**

  Performance design is necessary for RCS to satisfy given real-time constraints. Performance design includes performance analysis using simulation models and scheduling. A real-time operating system provides several mechanisms such as timer (set_timer, get_timer), task management (start_task, terminate_task, suspend_task), and priority control (change_priority). These mechanisms are used for the scheduling design to satisfy given real-time constraints.

  → Real-time Property and Nondeterminism

2.3 Difficulties in Developing RCS

Generally speaking the development (especially testing and debugging) of RCS is more difficult than that of TFS. The safety and reliability of RCS are very important as they are used in crucial systems such as power plants, chemical plants, and various computer embedded systems. As RCS become increasingly complex and distributed over computer networks complete testing of the safety of these programs becomes more difficult and the cost of testing and debugging becomes a heavy burden. The difficulty in testing and debugging RCS can be summarized as follows.

- **Concurrent thinking:** The concurrent model is fit to design static structures of systems. However, it is difficult to trace their dynamic behaviors concurrently because human thinking is essentially sequential. Concurrent thinking which is required for designing, testing and debugging RCS is very difficult for designers to achieve.
- **Data and timing variations:** In testing TFS, the designer has to consider only data variations. However, timing variations in addition to data variations have to be considered in testing RCS. Combination of data and timing variations makes testing very complicated.
- **Environment modeling:** Since a program of the reactive system cannot function without its environment, testing on the development machine requires an environment model and its simulator. The designer can test RCS with environment simulators, which are very useful in reducing testing costs. However, simulator construction requires additional programming costs, and it is ineffective to construct a handmade environment simulator for each RCS.
- **Lack of bug reappearance:** Unlike in TFS, bugs do not necessarily reappear in RCS. For example bugs which appear in the usual execution often disappear when using the debugging tool (it is called the *probe effect*). It makes debugging of RCS very difficult.
- **Task structuring:** As compared with module structuring in TFS, task structuring is more difficult because physical constraints (real-time constraints and constraints due to devices) must be considered in addition to logical module structuring.

3 Programming Environment for RCS Using Petri Nets and Temporal Logic

This section shows a basic concept and organization of the proposed software development process and its programming environment in this thesis. Then it is shown how the above difficulty is eased by them.

Formal Specification for RCS

In the proposed software development process, the formal specification takes up a position as a software design document (SDD) which is produced through the analysis and design phase. The formal specification is used for the formal verification and adjustment later.

Many formal specification languages are proposed. Some are declarative, and some are operational. However, specification of practical systems requires both a declarative one and an operational one. For example, it is impossible to describe all of the practical systems by temporal logic (declarative one). Conversely, it is impossible to describe *liveness properties* such as deadlock-free by Petri nets (operational one). An operational specification language is suitable for describing static and dynamic structures of the system. On the other hand, a declarative one is suitable for describing constraints of the system.

We adopt both Petri net and temporal logic as a formal specification language for RCS because they can complement each other

Design Document Reuse

The development process promotes and supports reuse of software design documents written by Petri nets and temporal logic. Reusable components are stored in the reusable component library. Since Petri nets provide a graphical representation of reusable components in the library, the designer can easily grasp these components. Temporal logic provides a formal framework to check whether reusable components work just as the designer designed.

Verification and Adjustment for RCS

The software design document can be verified by the formal method, that is, it is possible to verify whether the given Petri net satisfies the given temporal logic constraints.

Moreover, when the Petri net does not satisfy the temporal logic constraints, the Petri net can be automatically adjusted to satisfy them.

In particular, software reuse involves risk of bugs because the designer may not have an accurate understanding of reused components. So, verification and adjustment can complement software reuse.

Programming Language for RCS

The verified and adjusted software design documents are coded with the programming language in the implementation phase. The proposed development process also adopts Petri net as a programming language. In this case, implementation is expected to be smoother because both design document language and programming language are based on the same framework. However, since pure Petri net is insufficient as a practical programming language for RCS, it should be extended.

Design Methodology for RCS

Software design documents written by using Petri nets and temporal logic are very formal and strict. Therefore, it is very difficult to describe these documents directly from ambiguous and informal requirements in analysis and design phase. There are many design methods such as RSA/RSD and OOA/OOD. However, they do not use Petri nets as their design documents (charts). A Petri-net-oriented design method for RCS is required in the development process.

Software Development Process (Basic Concept)

Based on the above consideration, the proposed software development process is summarized as follows (Fig. 13).

1. Analyze and design a target system from informal specification according to the Petri-net-based design methodology.

2. Construct a software design document written by using Petri nets and temporal logic. When refining Petri nets, reusable components are available.

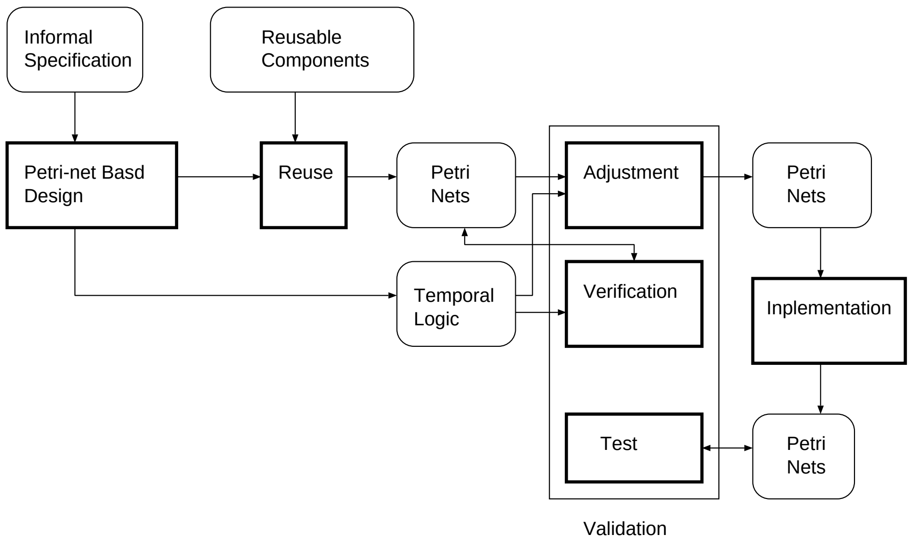


Figure 13. Proposed Software Development Process for RCS

3. Verify and adjust the design document.
4. Implement the design document and code it by using a Petri-net-based programming language.
5. Test the generated program.

Programming Environment for RCS

The programming environment supports the proposed software development process continuously. It includes

- Graphic editor for Petri nets
- Support tool for design and software reuse
- Verification and adjustment tool
- Compiler/Interpreter of Petri-net-based programming language
- Testing and debugging tools including program and environment simulators

How to Ease Difficulties in Developing RCS

The proposed software development process and its programming environment utilizing Petri nets and temporal logic can ease some difficulties of software development of reactive and concurrent systems as follows.

- **Petri Nets:** Petri nets provide a user-friendly graphical representation of reactive and concurrent systems and also their environments. Petri nets are adopted from first to last (i.e., from design to implementation) in the proposed software development process.

  → ease difficulties due to concurrent thinking, and environment modeling.

- **Temporal Logic:** Temporal logic provides formal methods of specification, verification, and adjustment about timing constraints of reactive and concurrent systems.

  → ease difficulties due to timing variations and lack of bug reappearance.

- **Methodology:** Design methodology featuring Petri nets and temporal logic provides support so that the designer can easily use these formal methods and design continuously from ambiguous requirement to program implementation.

  $\rightarrow$ ease difficulties in task structuring.

4 Summary

This chapter has shown a conceptual overview of the software development process for reactive and concurrent systems using Petri nets and temporal logic. In the following chapters, we explain the detail techniques required to realize this software development process.

Chapter 4

Specification, Verification, and Synthesis Using Petri Nets and Temporal Logic

Both Petri nets and temporal logic have been widely used to specify concurrent programs [Shatz 93]. Petri nets are appropriate to specify the behavioral structures of programs explicitly, while temporal logic is appropriate to specify the properties and constraints of programs. Since one can complement the other, using a combination of Petri nets and temporal logic is a highly promising approach to analyze, verify and synthesize concurrent programs. This fusion of Petri nets and temporal logic as specification language belongs to *dual-language approach*. For the purpose of automatic program verification and synthesis, the emptiness problem (i.e. whether a legal firing transition sequence satisfying a given temporal logic formula on a given Petri net exists) must be decidable. This chapter first reports a class to combine Petri nets and temporal logic as an infinite language and whose emptiness problem is decidable. Then, we apply these results to verification and synthesis of concurrent programs. Our verification method allows verifying several properties which cannot be covered by the traditional Petri net analysis, such as analysis of mutual exclusion and partial ordering of events. Our synthesis method can be used to modify an original concurrent program that is represented by a Petri net, to satisfy a given temporal specification.

1 Petri Nets and Temporal Logic as Specification Language

As mentioned at Chapter 1 (Section 2: Background), a specification of reactive and concurrent systems can be classified into *declarative approach* and *operational approach*. Petri net and temporal logic are typical specification languages based on operational approach and declarative approach, respectively.

The Petri net [Peterson 81, Murata 89] is widely accepted as a graphic and formal modeling tool, applicable to reactive and concurrent systems, and it is suited for modeling behavioral structures [Shatz 93]. However, Petri nets are deficient in their ability to describe declarative constraints of programs, which is a strong point of logic. On the other hand, temporal logic is successfully applied as a tool for the verification [Pnueli 77] and synthesis [Manna 84] of concurrent programs, and suited for specifying the constraints (declarative properties) of reactive and concurrent systems. Manna and Pnueli [Manna 92] classifies these properties into two disjoint classes.

- Safety Properties:
  A *safety property* claims that "something bad" does not happen.
- Liveness Properties:
  A *liveness property* claims that "something good" eventually happens.

Temporal logic can specify both safety property and liveness property. For example, prohibiting constraints, such as "once an error event occurs, a start event must not be activated" can be described explicitly by temporal logic, but it can only be described implicitly by Petri nets.

Petri net and temporal logic can complement each other as shown in Table 3. Therefore, it is useful to combine Petri nets and temporal logic as a specification language for analyzing, verifying and synthesizing concurrent programs. One can design concurrent programs operationally by Petri nets, while one can specify them declaratively by temporal logic. We consider *fusion* of temporal logic and Petri nets, which is called *dual-language approach* in [Felder 94].

Table 3. Comparison of Petri Nets and Temporal Logic

Petri Net	*Temporal Logic*
Operational(Executable)	Declarative(Unexecutable)
Suitable for Structure Description	Suitable for Constraint Description
Visual and Textual Representation	Textual Representation

Since a bounded Petri net has only finite states, expressive power of Petri nets combined with temporal logic is equivalent to one of finite state transition systems combined with temporal logic. Verification and adjustment using transition systems and temporal logic will be described in the subsequent chapters. We focus on unbounded Petri nets in this chapter.

Several classes [Cherkasova 87, Howell 88, Suzuki 89] have already been proposed in which unbounded Petri nets are combined with temporal logic. However, these classes are inadequate for automatic verification and synthesis because some are undecidable in regard to the emptiness problem (i.e. whether or not there exists a legal firing transition sequence satisfying a given temporal logic formula on a given Petri net), and some are decidable but their complexity is as enormous as the reachability problem. The decidability of the emptiness problem is inevitable for automatic verification and synthesis. In this chapter, we select a class combining Petri nets and propositional linear time temporal logic (PLTL), such that its expressive power is less than classes mentioned above, but its emptiness problem is decidable, and its complexity is the same as the coverability problem. In this class, a transition firing sequence corresponds to a model of the temporal logic formula, and it is possible to combine Petri nets and temporal logic as an infinite Petri net language. Infinite Petri net languages were well investigated by Valk [Valk 83, Valk 85]. The following results will be shown in this chapter using infinite Petri net language techniques:

1. It is decidable whether or not a Petri net satisfies a propositional temporal logic specification.
2. For a given Petri net N and propositional temporal logic specification f, the new Petri net N' can be constructed by modifying N, such that N' satisfies f.

These results are applied to verification and synthesis for concurrent programs in Sections 2 and 3, respectively.

1.1 How to Fuse Petri Nets and Temporal Logic

There are several ways to combine a Petri net with temporal logic. The key point in combining is what the atomic proposition in temporal logic corresponds to in Petri nets. There are some possible correspondences between atomic propositions in PLTL and Petri net properties as follows.

(a) an atomic proposition $mk(p)$ is true iff place p has at least one token.

(b) an atomic proposition $ge(p, c)$ is true iff place p has at least c tokens. [6]

(c) an atomic proposition $en(t)$ is true iff a transition t is enabled.

(d) an atomic proposition $fi(t)$ is true iff a transition t just fires. [7]

For these correspondences, several research results are presented as shown in Table 4. It can be seen that the emptiness problem becomes undecidable in some Petri nets combined with temporal logic. Some are decidable, but are limited to bounded Petri nets or restricted PLTL. Here, the emptiness problem is roughly defined as to decide whether, for a given Petri net N and a given temporal logic specification

[6] (a) is a special case of (b), i.e. $mk(p) = ge(p, 1)$.
[7] Note that $fi(t) \supset en(t)$ always holds.

f, there exists a legal firing transition sequence on N satisfying f. Our purpose is to select an general Petri net class combined with general PLTL in which the emptiness problem is decidable. The reason is that decidability is necessary for automatic program verification and synthesis, and unboundedness of the general Petri net is necessary for modeling asynchronous communication in concurrent programs.

Table 4. Several Combinations of Petri Nets and Temporal Logic

Paper	Type	Petri net	Emptiness Problem
[Katai 82]	a	safe	decidable
[Cherkasova 87]	b,c,d	general	undecidable
[Howell 88]	a,c,d	general	undecidable
[Suzuki 89]	b,c,d	general	undecidable
			decidable for restricted PLTL
[Uchihira 90a]	d	bounded	decidable

Here, we adopt only d-type correspondence and combine the Petri net N and PLTL formula f as a formal language. A labeling function $h : T \to Prop \cup \{\varepsilon\}$ is used to map transitions $t \in T$ of N to atomic propositions $fi(t) \in Prop$ (i.e., $h(t) = fi(t)$). In addition, all transitions of N do not necessarily correspond to atomic propositions. Some transitions may be invisible to a user who describes temporal logic specifications (i.e. $h(t) = \varepsilon$ for a invisible transition t). We are now going to generally define a new formal language from N, f, and h.

Definition 13 ($L(N, f, h)$) *$N = (P, T, w, m_0)$ is a Petri net, and f is a PLTL formula which consists of atomic propositions Prop, and h is a labeling function such that $h : T \to Prop \cup \{\varepsilon\}$. We define $L(N, f, h) \overset{def}{=} L_\omega(N, h) \cap L_s(f)$, where $L_\omega(N, h)$ is an infinite language generated from a labeled Petri net (N, h), and $L_s(f)$ is an infinite language generated from f under the single event condition [8] . $L(N, L, h) \neq \emptyset$ means that there exists a legal firing transition sequence satisfying f on N.*

1.2 Example of Specification

A typical example of reactive and concurrent systems, mutual exclusion, is considered. In this example, a behavioral structure of the system is given by Petri net N shown in Fig. 14, and declarative constraints are given by the following temporal logic formula.

- d-type

$$f = \Box\Diamond fi(t_{11}) \wedge \Box\Diamond fi(t_{21}) \wedge \Box(fi(t_{11}) \supset \bigcirc(\neg fi(t_{21})\, U\, fi(t_{12}))) \wedge \Box(fi(t_{21}) \supset \bigcirc(\neg(fi(t_{11}))\, U\, fi(t_{22})))$$

- a-type

$$f = \Box\Diamond mk(CS_1) \wedge \Box\Diamond mk(CS_2) \wedge \Box\neg(mk(CS_1) \wedge mk(CS_2))$$

Here, CS and NCS mean "critical section" and "non-critical section", respectively. Both formulas mean that this Petri net is deadlock-free and the mutual exclusion of critical sections CS_1 and CS_2 is preserved.

1.3 Theoretical Results

Theorem 1 (Decidability of Emptiness Problem of $L(N, f, h)$**)** *The emptiness problem of $L(N, f, h)$ (i.e. $L(N, f, h)$ is empty or not) is decidable for given N, f, h.*

Proof. It is sufficient to prove that the emptiness problem of $L_\omega(N, h) \cap L_s(f)$ is decidable. To begin with, the following procedures are provided which constructs an extended coverability graph G from N, h, and $L_s(f)$.

[8] $L_\omega(N, h)$ and $L_s(f)$ are defined in Chapter 2.

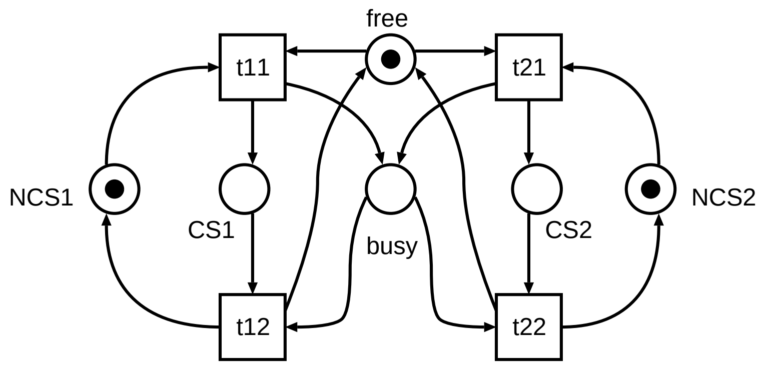


Figure 14. Petri Net N Representing Mutual Exclusion

Main Procedure

1. A *Büchi sequential automaton* $A_f = (Prop, S, \rho, s_0, F)$ accepting $L_s(f)$ is constructed.
2. Then, construct an extended coverability graph G from $N = (P, T, w, m_0)$, h and A_f. G is a labeled directed graph. Each node x of G is represented as a k+2-tuple $x = (x_1, ..., x_k, s, f)$ where $|P| = k, x_i \in \{0, 1, ...\} \cup \{\omega\}(1 \le i \le k)$, $s \in S$, $f \in \{0(\text{normal node}), 1(\text{designated node})\}$. Each edge $e = (x, x')$ is labeled with an element of T. A transition $t \in T$ is called to be *enabled* in x if t is enabled at a marking $(x_1, ..., x_k)$ and $\rho(s, h(t)) \ne \emptyset$, and t is called to be *local enabled* if t is enabled in the marking and $h(t) = \varepsilon$. G is constructed as follows:
   (a) Start with a graph G containing only an initial node $x_{init} = (x_{01}, ..., x_{0k}, s_0, f)$ where $m_0 = (x_{01}, ..., x_{0k})$, s_0 is an initial state of A_f, and $f = 1$ if $s_0 \in F$, otherwise $f = 0$. Let x_{init} be an unapplied node.
   (b) Repeatedly apply the following Graph Addition Procedure to the new (unapplied) nodes of G until all nodes of G have been applied.

Graph Addition Procedure

1. Let x be a given node with $x = (x_1, .., x_k, s, f)$. Create new node candidates $x' = (x'_1, ..., x'_k, s', f')$ from x according to the following (a)-(d) for all enabled transitions t at x and all $s' \in \rho(s, h(t))$. Also create new node candidates $x' = (x'_1, ..., x'_k, s, 0)$ according to (a)-(c) for all local enabled transitions t:
   (a) $x'_i = \omega$ if $x_i = \omega$ $(1 \le i \le k)$.
   (b) If there is a node $y = (y_1, ..., y_k, s', f_y)$ on some path from x_{init} to x (that is an ancestor of x) such that $y_j \le x_j - w(p_j, t) + w(t, p_j)$ for all j $(1 \le j \le k)$ and $y_i < x_i - w(p_i, t) + w(t, p_i)$ for some i, then $x'_i = \omega$.
   (c) For the other i, $x'_i = x_i - w(p_i, t) + w(t, p_i)$.
   (d) $f = 1$ if $s' \in F$, otherwise $f = 0$.
2. If x' is new in G, that is, G doesn't have the same node, then add a new node x' and a new edge $e = (x, x')$ labeled with t. Otherwise add only a new edge $e = (x, x')$ labeled with t.

The above procedure always terminates because G is finite in the same way of the normal coverability graph of a Petri net. Next, we prove the following claim.

Claim: $L_\omega(N, h) \cap L_s(f) \ne \emptyset$ iff there exists a cycle $c = n_0 n_1 ... n_k n_0$ on G such that n_0 is a designated node (i.e. $f = 1$) and $\Delta(\theta) \ge 0$ where $\theta = t_1 ... t_{k+1}$ is a transition sequence over c (i.e. t_i is a label of $e(n_{i-1}, n_i)$).

Note that θ is not necessary to be legal. The above claim follows directly from the result (Theorem 3.11) of Valk and Jantzen [Valk 85]. Furthermore, it is decidable whether or not there exists such a cycle as follows: For each designated node n_0, a set of all transition sequences θ forming cycles on G passing through n_0, can be represented by a regular expression R (e.g. $R = (t_1t_2)^* + (t_1(t_3 + t_4)^*t_5)^*$). We want to decide if there are some $\theta \in R$ such that $\Delta(\theta) \geq 0$. For this purpose, we can regard R as commutative. Therefore, R can be expressed as a finite sum of terms of the shape $\theta_0\theta_1^*\theta_2^*...\theta_n^*$ such that $\theta_i \in T^*(0 \leq i \leq n)$, using the decomposition rules ($AB = BA$, $A^*B^* = (AB)^*(A^* + B^*)$, $(A + B)^* = A^*B^*$, $(A^*B)^* = \varepsilon + A^*B^*B$), which are described in the Conway's book [Conway 71]. For each $\theta_0\theta_1^*\theta_2^*...\theta_n^*$, we can effectively decide whether $\Delta(\theta_0\theta_1^{\alpha_1}\theta_2^{\alpha_2}...\theta_n^{\alpha_n}) = \Delta(\theta_0) + \alpha_1\Delta(\theta_1) + \alpha_2\Delta(\theta_2) + ...\alpha_n\Delta(\theta_n) \geq 0$ for some $\alpha_1, \alpha_2, ..., \alpha_n \geq 0$, by using linear programming.

While Valk and Jantzen [Valk 85] showed that it is decidable whether there is a legal firing sequence on a Petri net satisfying various fairness constraints, this theorem shows that it is decidable whether there is a legal firing sequence on a Petri net satisfying temporal logic constraints. It is also possible to prove this theorem by showing Petri net + temporal logic constraints to be equivalent to Petri net + some fairness constraints.

When $L(N, f, h)$ is nonempty, it is very important to find a concrete sequence accepted by $L(N, f, h)$ for the sake of program synthesis.

Theorem 2 (Construction of Legal Firing Sequence) *If $L(N, f, h)$ is nonempty, we can construct a deterministic legal firing sequence $\theta_0\theta_c^\omega$ on N such that $h(\theta_0\theta_c^\omega) \in L(N, f, h)$.*

Proof sketch. First, we redefine a weight function Δ as $\Delta(e) = \Delta(\text{transition of edge } e)$ for edges of G. If $L(N, f, h)$ is nonempty, there exists a cycle $c = e_0e_2...e_k$ on G where $e_0 = (x_0, x_1), ..., e_k = (x_{k-1}, x_0)$, $\Delta(c) \geq 0$, and x_0 is a designated node from Theorem 1. Remark that c might not be legal. We will show cycle c can be constructed, and we can also construct a path p from the initial node x_{init} to the designated node x_0, whose transition sequence is legal and a marking in x_0 (let it m_{x_0}) is so large that c becomes legal.

- **Construction of cycle** c

  In the same way as the proof of Theorem 1, we can compute $\alpha_1, \alpha_2, ..., \alpha_n \geq 0$, such that $\Delta(c) = \Delta(e'_0) + \alpha_1\Delta(e'_1) + \alpha_2\Delta(e'_2) + ...\alpha_n\Delta(e'_n) \geq 0$ where e'_i may not be adjacent to e'_{i+1}. Each α_i means how many times the edge e'_i appears in c. Therefore, the construction of c can be reduced to the construction of Euler cycle c such that each edge e'_i appears α_i times in c.

- **Construction of path** p

  We can form the path into $p = p_0c_1^{\alpha_1}p_1c_2^{\alpha_2}...c_n^{\alpha_n}p_n$ where c_i means a cycle in which some place's marking changes into ω on G (we call it i-th ω place). We would like to compute $\alpha_1, \alpha_2, ..., \alpha_n \geq 0$ such that $\Delta(p_0c_1^{\alpha_1}p_1c_2^{\alpha_2}...c_n^{\alpha_n}p_n) \geq m_{x_0}$. Here, the problem is not so easy because p must be legal. The following recurrent formulas must be solved:

  (0) $M_0 = m_0 + \Delta(p_0)$, and a path p_0 from marking m_0 must be legal,

  (i) $M_i = M_{i-1} + \alpha_i\Delta(c_i) + \Delta(p_i)$, and a path $c_i^{\alpha_i}p_i$ from marking M_{i-1} must be legal,

  (n) $M_n \geq m_{x_0}$.

  We point out that these formulas can be solved backward from the n-th formula, that is, we can compute α_i backward independently from $\alpha_1, \alpha_2, ..., \alpha_{i-1}$, because we can ignore j-th ω place for all $j < i$.

After all, a deterministic legal transition sequence $\theta_0\theta_c^\omega$ such that $h(\theta_0\theta_c^\omega) \in L(N, f, h)$ is directly produced from pc^ω.

2 Concurrent Program Verification

Consider concurrent program verification focusing on behavioral properties. After retracting the basic behavioral structures represented by Petri nets from concurrent programs, it is possible to analyze the behavioral properties of programs. This verification means to check whether or not a given Petri net satisfies a given specification. Temporal logic is adopted as a specification language where atomic propositions correspond to transition firing as described in the previous chapter. Only the infinite Petri net language $L_\omega(N, h)$ is considered there, which does not care for finite behaviors of N including deadlocks. Therefore, Petri net N is extended to Petri net N_ω which is made deadlock-free by adding a visible dummy transition nop (no operation) in Fig. 15.

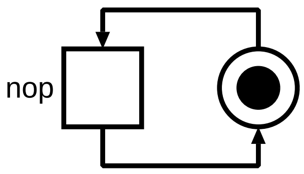


Figure 15. nop (no operation)

It will be shown how to verify whether a given concurrent program (N_ω) meets a given specification (f). We assume that all transition are visible and atomic propositions of f are identified with transitions. In this case, a labeling function to map transition to atomic propositions is an identity function e. To verify that the program meets the specifications, it suffices to check $L_s(f) \supset L_\omega(N_\omega, e)$, which means each of all possible computations in Petri net N_ω is a model of PLTL formula f.

Definition 14 *A deadlock-free Petri net N_ω satisfies the temporal logic specification f with the single event condition iff $L_s(f) \supset L_\omega(N_\omega, e)$. Deciding whether N_ω satisfies f is called a verification problem.*

Theorem 3 *The verification problem is decidable.*

Proof. From Chapter 2 Lemma 2, the verification problem ($L_s(f) \supset L_\omega(N_\omega, e)$) can be reduced to the emptiness problem ($L_s(\neg f) \cap L_\omega(N_\omega, e) = L(N_\omega, \neg f, e) = \emptyset$). Therefore, it is decidable from Theorem.

It will now be made clear what the inputs and outputs of this verification are:

INPUT: Concurrent program structure (represented by Petri net N_ω).

INPUT: Specification (represented by temporal logic f),

OUTPUT: Yes / No,

where "yes" means that the program satisfies the specifications, and "no" means it does not.

It is significant to analyze what is possible and what is not in this verification method as follows:

What is possible to verify

- Mutual exclusion

  ex. Intervals $[t_1, t_2]$ and $[t_3, t_4]$ between two transitions do not overlap each other:

$$\Box(t_1 \supset \bigcirc(\neg(t_3 \vee t_1)\ U\ t_2)) \wedge \Box(t_3 \supset \bigcirc(\neg(t_1 \vee t_3)\ U\ t_4))$$

  Note: Though we cannot directly specify tokens of places, in the case of Fig. 16(a) an interval $[t_1, t_2]$ can specify that place p has a token in the interval.

- Partial ordering among transition firing

  ex. Transitions t_1 and t_2 fire in turn:

$$\Box(t_1 \supset \bigcirc(\neg t_1\ U\ t_2)) \wedge \Box(t_2 \supset \bigcirc(\neg t_2\ U\ t_1))$$

- Firing prohibition

  ex. Once t_1 fires, t_2 will never fire.

$$\Box(t_1 \supset \bigcirc\Box\neg t_2)$$

- Deadlock inevitability

  ex. Transition t will eventually fall into deadlock.

$$\Diamond\Box\neg t$$

- Boundedness and safeness property It can be verified by introducing dummy transitions. ex. $\Diamond\Box\neg d$ to verify whether place p is bounded in Petri net with a dummy transition d (Fig. 16(b))

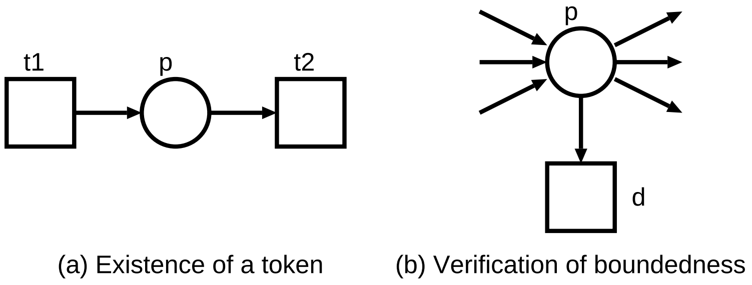


Figure 16. What is Possible to Verify

What is impossible to verify

- Number of tokens

  It is impossible to generally verify the number of tokens in places, which could be used to specify reachability property.

- Possibility of deadlock (liveness property)

  This arises from the introduction of nop.

Consideration for complexity gives a clear interpretation to these possibility and impossibility of verification. When PLTL formula is small enough compared with Petri net (in fact all example formulas mentioned above are small), complexity of our verification method has the almost same order as the coverability problem. Therefore, our method cannot verify in itself the reachability problem and liveness property, of which complexity is far larger than the coverability problem.

Verification Example As a simple example, verifying a concurrent program, let's consider a mutual exclusion problem containing unbounded buffers. A target Petri net N_ω is illustrated in Fig. 17, where places p_4 and p_5 are unbounded buffers (this Petri net is deadlock free itself, therefore we ignore nop for a simple explanation). Specification f is given such that intervals $[t_1, t_2]$ and $[t_3, t_4]$ satisfy a mutual exclusion condition, as follows:

$$f \overset{def}{=} \Box(t_1 \supset \bigcirc(\neg t_3\ U\ t_2)) \wedge \Box(t_3 \supset \bigcirc(\neg t_1\ U\ t_4))$$

Büchi sequential automaton $A_{\neg f} = (\{t_1, t_2, t_3, t_4\}, \{s_0, s_1, s_2, s_3\}, \rho, s_0, \{s_3\})$ is shown in Fig. 18. The extended coverability graph G can be generated from $A_{\neg f}$ and N_ω (Fig. 19). In G, no designated node exists. That means $L(N_\omega, \neg f, e) = \emptyset$ from Theorem 1. We conclude N_ω satisfies f.

3 Concurrent Program Synthesis

It is not easy for an ordinary programmer to realize a correct synchronization in concurrent programs, and it requires tremendous debugging efforts. This section provides a method to synthesize a concurrent program automatically with reusable components by program tuning. The goal programs are synthesized to satisfy the given specification by tuning up reused programs that are represented by Petri nets. We also emphasize that our method adopts a compositional way to synthesize. It is necessary for two reasons:

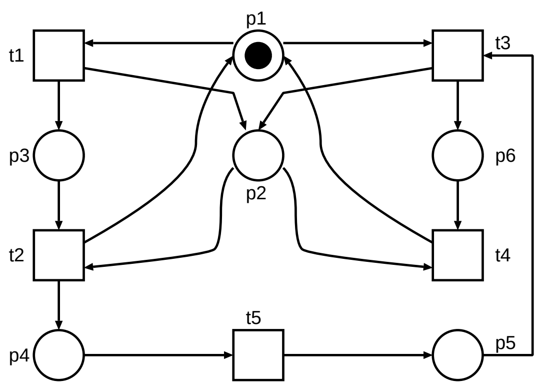


Figure 17. N_ω: Mutual Exclusion with an unbounded buffer

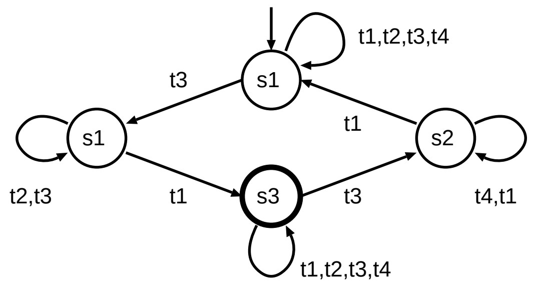


Figure 18. $A_{\neg f}$

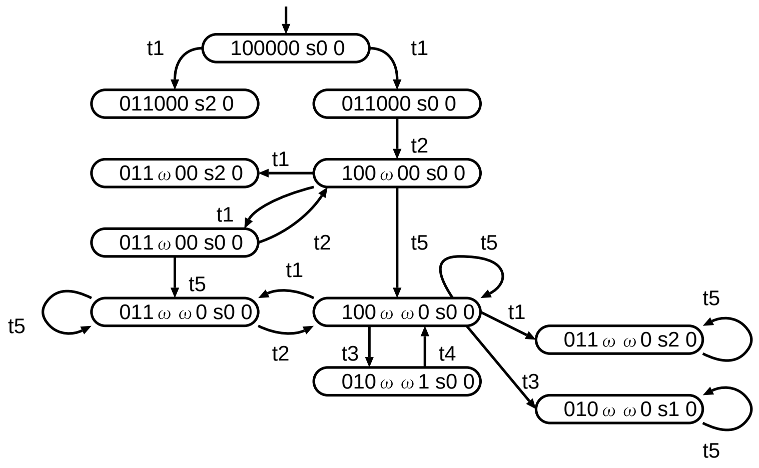


Figure 19. G: Extended Coverability Graph

- Reusable software itself is composed compositionally in ordinary software, and
- global synthesis of a large-scale program requires huge, and therefore impractical, computing power.

The model building techniques in Theorem 2 are used in this synthesis method.

3.1 Composition of Petri Nets

We also introduce a composition operator "$|_L$" which plays an important role in the synthesis method.

Definition 15 (Composition of labeled Petri nets) *For given labeled Petri nets $LN_1 = (N_1, h_1)$, $LN_2 = (N_2, h_2)$, and a given label set $L \subset \Sigma$ where $N_1 = (P_1, T_1, w_1, m_{01})$, $N_2 = (P_2, T_2, w_2, m_{02})$, $P_1 \cap P_2 = \emptyset$, and $T_1 \cap T_2 = \emptyset$, a labeled Petri net $(N, h) = LN_1 \mid_L LN_2$, which is called a composition of LN_1 and LN_2 with L, is defined as follows.*

$N = (P, T, w, m_0)$, where

- $P = P_1 \cup P_2$,
- $T = T'_1 \cup T'_2 \cup T'$ *such that*
  - $T'_1 = \{t \in T_1 | h_1(t) \notin L\}$,
  - $T'_2 = \{t \in T_2 | h_2(t) \notin L\}$,
  - $T' = \{t_{ij} \mid h_1(t_i) = h_2(t_j) \in L\}$,
- $w(p, t) =$

$$\begin{cases} w_1(p,t) \text{ for } t \in T'_1 \\ w_2(p,t) \text{ for } t \in T'_2 \\ w_1(p,t_i) + w_2(p,t_j) \text{ for } t = t_{ij} \in T' \end{cases}$$

  $w(t, p)$ is also defined similarly,
- $m_0(p) = m_{01}(p) + m_{02}(p)$ *for all $p \in P$.*

Example 4 *We show an example of a composition $(N_1, h_1) \mid_L (N_2, h_2)$ in Fig. 20. In this example, $h_1(t_1) = b, h_2(t_4) = c, h_1(t_2) = h_2(t_3) = a$, and $L = \{a\}$.*

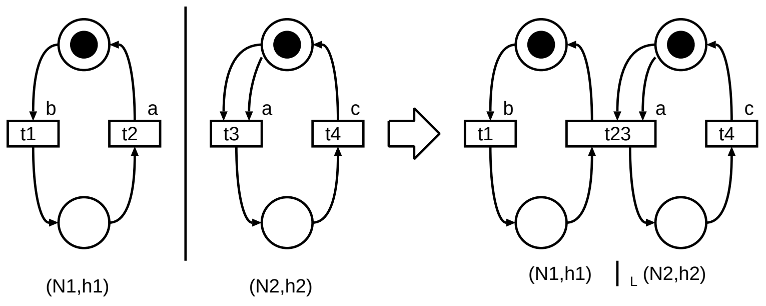


Figure 20. Composition of Petri Nets

Lemma 4 *Let LN, LN_1,LN_2 be labeled Petri nets and $L \subset \Sigma$ be a set of labels. When $LN = LN_1 \mid_L LN_2$, $L(LN)/L = L(LN_1)/L \cap L(LN_2)/L$, $L_\omega(LN)/L = L_\omega(LN_1)/L \cap L_\omega(LN_2)/L$, and $L_{\Delta\omega}(LN)/L \supset L_{\Delta\omega}(LN_1)/L \cap L_{\Delta\omega}(LN_2)/L$. Here, $L(LN)$ represents $L(N, h)$ such that $LN = (N, h)$.*

Proof. $L(LN)/L = L(LN_1)/L \cap L(LN_2)/L$ and $L_\omega(LN)/L = L_\omega(LN_1)/L \cap L_\omega(LN_2)/L$ are clear. We prove $L_{\Delta\omega}(LN)/L \supset L_{\Delta\omega}(LN_1)/L \cap L_{\Delta\omega}(LN_2)/L$ by showing a example $\eta \in L_{\Delta\omega}(LN)/L$ and $\eta \notin L_{\Delta\omega}(LN_1)/L \cap L_{\Delta\omega}(LN_2)/L$ (Fig.21). In $LN = LN_1 \mid_{\{c_1,c_2\}} LN_2$, a firing sequence $\theta = t_1 t_5$ falls into deadlock. Therefore, $h(\theta) = \Delta^\omega \in L_{\Delta\omega}(LN)/L$, but $h(\theta) \notin L_{\Delta\omega}(LN_1)/L \cap L_{\Delta\omega}(LN_2)/L$.

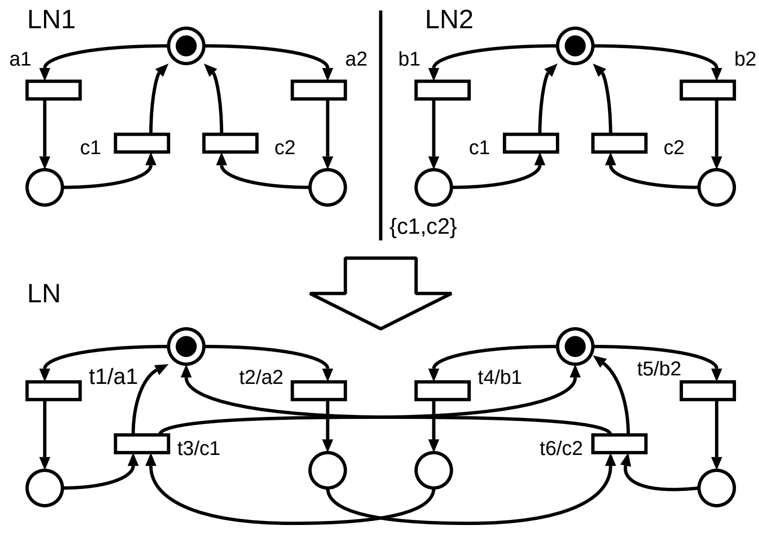


Figure 21. Counter Example of $L_{\Delta\omega}(LN)/L = L_{\Delta\omega}(LN_1)/L \cap L_{\Delta\omega}(LN_2)/L$

3.2 Petri Net Synthesis

Definition 16 *$L^{fair}_{\Delta\omega}(N,h) \subset L_{\Delta\omega}(N,h)$ is defined as*

$$L^{fair}_{\Delta\omega}(N,h) \overset{def}{=} \{h(\theta) \in \Sigma^\omega \cup \Sigma^*\Delta^\omega \mid \theta \in F_\omega(N) \text{ under the visibility-fairness condition}\}$$

where the visibility-fairness condition means whenever some visible transitions are infinitely enabled, then one of them will eventually fire. Here, $L_\omega(N,h) \subset L^{fair}_{\Delta\omega}(N,h)$

Definition 17 *A Petri net $N = (P,T,w,m_0)$ is deadlock-free iff there is at least one enabled transition for every reachable marking. A labeled Petri net (N,h) is visibility-starvation-free iff $L^{fair}_{\Delta\omega}(N,h) = L_\omega(N,h)$, that is, there is no infinite loop of invisible transitions (which looks like deadlock for the outside) under the visibility-fairness condition.*

Now, we will show how to synthesize a new Petri net which satisfies temporal logic specifications and is deadlock-free and visibility-starvation-free, by tuning up the original net (i.e. adding some places, transitions, and arcs to the original net).

Theorem 4 (Petri Net Synthesis) *If $\sigma_0\sigma_c^\omega \in L_\omega(N,h)$, a labeled Petri net (N',h') can be constructed by adding some places, transitions, and arcs to N such that N' is deadlock-free, (N',h') is visibility-starvation-free, and $L_\omega(N',h') = \sigma_0\sigma_c^\omega$.*

Proof. It is easy to construct a labeled Petri net (N_s,e) such that $L(N_s,e) = \{s_0 s_c^*\}$. Then, make a composed Petri net $(N_c,h_c) = (N,h) |_\Sigma (N_s,e)$. Here, $L_\omega(N_c,h_c) = L_\omega(N,h) \cap L_\omega(N_s,e) = \{s_0 s_c^\omega\}$ from Lemma 4. Finally, we can construct (N',h') by tuning up (N_c,h_c) according to the Valk and Jantzen's tuning method [Valk 85] (cf. Appendix I) such that N' is deadlock-free, and $L^{fair}_{\Delta\omega}(N',h') = L_\omega(N',h') = s_0 s_c^\omega$.

Corollary 1 *Let (N,h) be a labeled Petri net and f be a PLTL formula. If $L(N,f,h) \neq \emptyset$, a labeled Petri net (N',h) can be constructed by adding some places, transitions, and arcs to N such that N' is deadlock free, (N',h') is visibility-starvation-free, and $L^{fair}_{\Delta\omega}(N',h') \subset L(N,f,h) \subset L_s(f)$.*

From now on, we abbreviate $fi(t)$ to just t for simplicity. In this case, a labeling function h of d-type forms $h = e/L$ where L is a set of visible transitions (i.e. atomic propositions).

Example 5 (Petri Net Synthesis) *A Petri net N is given in Fig. 22(a), and $h = e/\{t_0, t_1, t_2\}$ is a labeling function of N. And $\theta_0\theta_c^\omega \in L_\omega(N,h)$ is given where $\theta_0 = t_0$ and $\theta_c = t_1t_2$. First, N_θ is constructed (Fig. 22(b)) such that $L(N_\theta, e) = \{\theta_0\theta_c^*\}$. Then, $(N_c, h) = (N, h) \mid_\Sigma (N, e)$ is composed (Fig. 22(c)). Finally, N' is constructed as shown in Fig. 22(d). Here, $L_{\Delta\omega}^{fair}(N', h) = L_\omega(N', h) = \{\theta_0\theta_c^\omega\}$, and N' is tuned up to be deadlock-free while N_c is not.*

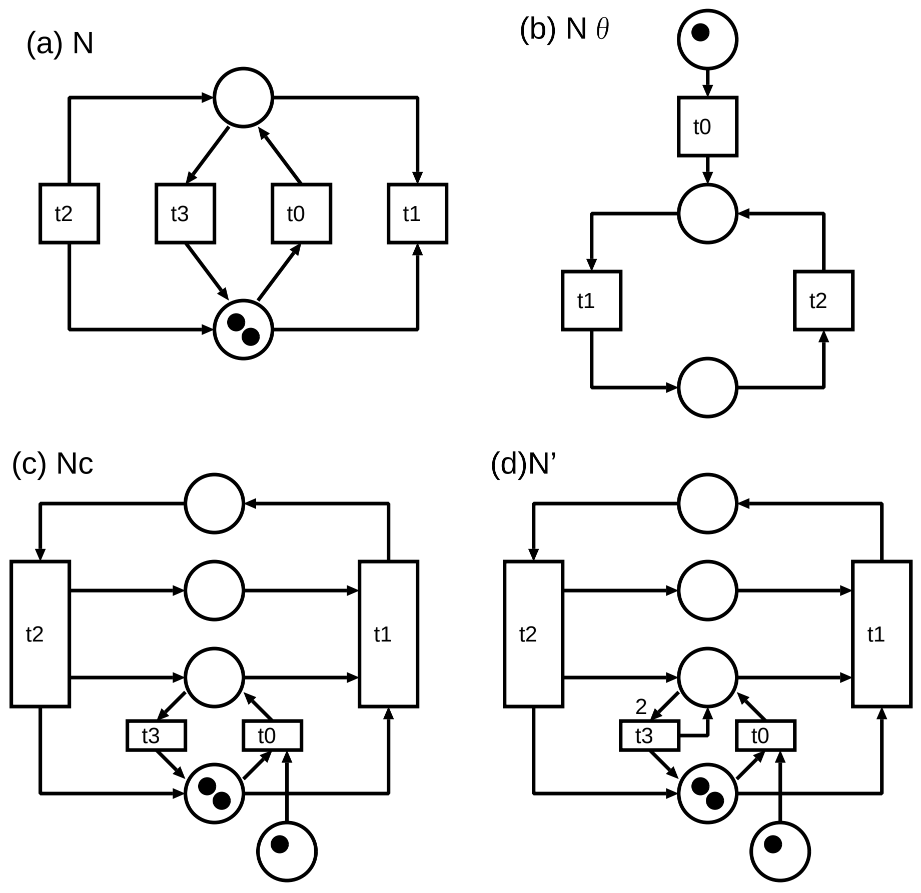


Figure 22. Example of Petri Net Synthesis

3.3 Concurrent Program Structure

It is assumed that a target program consists of one *controller* (main controller) and several *agents* (device controller) which control devices locally. While the controller controls each agent sequentially, the agent is independent from other agents and they can run concurrently with each other. This structure (*hierarchical synchronization supervisor*, ref. Chapter 3) is very natural in some domains, such as robot control systems and plant control systems. An example (Example 6) is shown in Fig. 23.

The controller and the agent i communicate with each other by a set of synchronous communication channels C_i, like CCS [Milner 89]. It is assumed that a raw controller and raw agents have already been constructed from reusable software components up to this step. Here, the raw controller is represented by Petri net $N_c = (P_c, T_c, w_c, m_{c0})$, and the raw agent i is represent by $N_{a_i} = (P_{a_i}, T_{a_i}, w_{a_i}, m_{a_{i0}})$. Communication channels C_i between N_c and N_{a_i} are defined as $C_i = \{h_i(t) \in T_c \mid t \in T_{a_i}\}$ where $hi : T_{a_i} \rightarrow T_c \cup \{\varepsilon\}$ is a labeling function (called "channel function") to connect N_{a_i}'s transitions with N_c's transitions. If $t \in T_{a_i}$ and $h_i(t) = \varepsilon$, t is a internal and invisible transition which is not connected with T_c. In case of Example 6, a controller and an agent may be represented as shown in Fig. 24, and channel functions are defined as $h_i(start) = start_i$, $h_i(end) = end_i$, $h_i(overflow) = \varepsilon$ for each i.

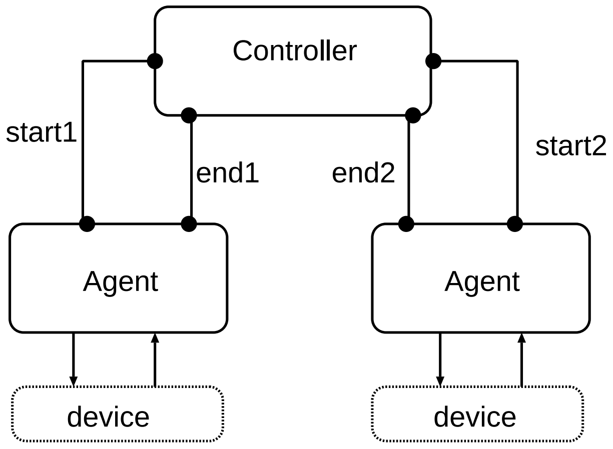


Figure 23. Concurrent Program Structure

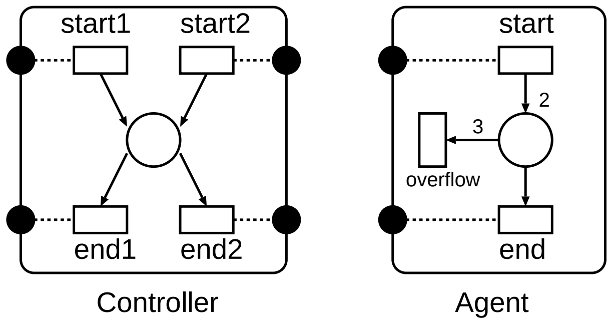


Figure 24. Original Controller and Agents

3.4 Temporal Logic Specification

The user specifies several constraints by a PLTL formula f with a set of atomic propositions $Prop \subset T_c$ so that the controller satisfies f cooperating with all agents.

Example 6

- $Prop = T_c = \{start_1, end_1, start_2, end_2\}$
- $f = \Box(start_1 \supset \bigcirc(\neg start_2 \; U \; end_1)) \wedge \Box(start_2 \supset \bigcirc(\neg start_1 \; U \; end_2))$

The formula f above means that once *Agent 1* starts, *Agent 2* never starts until *Agent 1* ends. Also when *Agent 2* starts, *Agent 1* never starts until *Agent 2* ends.

Here, a concurrent program synthesis means to tune up reusable components to satisfy this specification (constraint). To start with, it is made clear what the inputs and outputs are:

INPUT: Specification f (written by PLTL)

INPUT: Reused Programs
One raw Controller and several raw Agents and Channels (represented by Petri nets $N_c, N_{a_1}, N_{a_2}, ..., N_{a_k}$, and channel functions $h_1, h_2, ..., h_k$)

OUTPUT: Synthesized Programs
One Controller and several Agents and Channels (represented by Petri nets $N'_c, N'_{a_1}, N'_{a_2}, ..., N'_{a_k}$, and channel functions $h'_1, h'_2, ..., h'_k$)

The proposed program synthesis method consists of two procedures. First, we show the controller synthesis procedure and then the agent synthesis procedure.

3.5 Controller Synthesis

This controller synthesis procedure consists of the following four steps:

Step 1: Each Petri net N_{ai} of agent i is reduced as far as possible [Lee 85] into $N^r_{a_i}$ such that $L_\omega(N_{a_i}, h_1) = L_\omega(N^r_{a_i}, h_1)$.

Step 2: Make a composed Petri net $(N, h) = (N_c, e) \mid_{C_1} (N^r_{a_1}, h_1) \mid_{C_2} \cdots \mid_{C_k} (N^r_{a_k}, h_k)$. We abbreviate this composition to $Sync(N_c, N^r_{a_1}, N^r_{a_2}, ..., N^r_{a_k}, h_1, h_2, ..., h_k)$, since it means synchronization of Processes with channels.

Step 3: Construct an infinite firing sequence $\theta = \theta_0\theta_c^\omega$ on N such that $h/Prop(\theta_0\theta_c^\omega) \in L(N, f, h/Prop)$ from Theorem 2.

Step 4: Construct a deterministic Petri net N'_c such that $L(N'_c, e) = \{\theta_0\theta_c^*/T_c\}$.

N'_c is a Petri net of the synthesized controller, that is a deterministic sequential program. In case of Example 6, N'_c is synthesized from a transition sequence $\theta = (start_1 end_1 start_2 end_2)^\omega$ which satisfies specification f, as shown in Fig. 25.

3.6 Agent Synthesis

For each agent, we can construct a tuned agent Petri net $N'_{a_i} = (P'_{a_i}, T'_{a_i}, w'_{a_i}, m'_{a_i 0})$ and a labeling function $h'_i : T'_{a_i} \to C_i$ from $N^r_{a_i}$, and $\theta_0\theta_c^\omega$ such that N'_{a_i} is deadlock-free, (N'_{a_i}, h'_i) is visibility-starvation-free, and $L_\omega(N'_{a_i}, h'_i) = \{\theta_0\theta_c^\omega/C_i\}$, using Theorem 4. Fig. 26 shows a composition $(N^r_{a_i}, h_i) \mid_{c_i} (N_\theta, e)$ where $L_\omega(N_\theta, e) = \{(start_i end_i)^*\}$, and a synthesized agent in Example 6.

Note that the synthesized controller is a deterministic sequential program while the tuned agents can be nondeterministic concurrent programs. Here, we must assume the visibility-fairness condition for each agent. After all, the controller N'_c and the agents N'_{a_i} can run concurrently with synchronous communication by channels, of which structure is represented by $Sync(N'_c, N'_{a_1}, N'_{a_2}, ..., N'_{a_k}, h'_1, h'_2, ..., h'_k)$. The following theory assures that this composed concurrent program satisfies a given temporal logic specification.

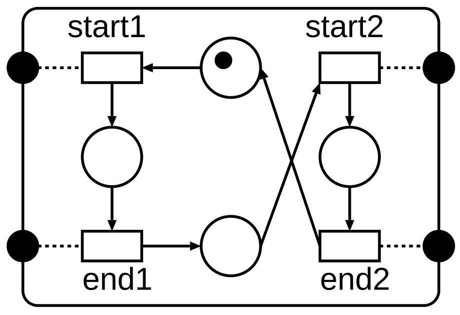


Figure 25. Synthesized Controller N'_c

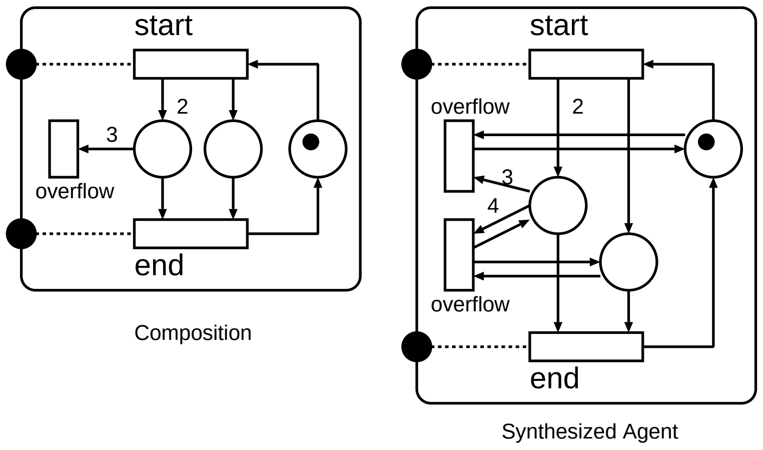


Figure 26. Composition $(N^r_{a_i}, h_i) \mid_{c_i} (N_\theta, e)$ and Synthesized Agents N'_{a_i}

Theorem 5 *If labeled Petri nets $(N'_c, e), (N'_{a1}, h'_1), (N'_{a2}, h'_2), ..., (N'_{a_k}, h'_k)$ are synthesized from Petri nets $N_c, N_{a_1}, N_{a_2}, ..., N_{a_k}$, channel functions $h_1, h_2, ..., h_k$, and a PLTL formula f with a set of atomic propositions Prop, according to the above synthesis method, then*

- *a composed labeled Petri net $(N', h') = Sync(N'_c, N'_{a_1}, N'_{a_2}, ..., N'_{a_k}, h'_1, h'_2, ..., h'_k)$ is deadlock free,*
- *(N', h') is visibility-starvation-free, and*
- *$L_\omega(N', h'/Prop) \subset L(N, f, h/Prop) \subset L_s(f)$, under the visibility-fairness condition where $(N, h) = Sync(N'_c, N'_{a_1}, N'_{a_2}, ..., N'_{a_k}, h'_1, h'_2, ..., h'_k)$.*

Proof. It is followed from our method utilizing previous theorems.

The main drawback in this synthesis is that a synthesized controller is deterministic. The controller is serialized by a deterministic firing sequence, while agents are non-deterministic and run concurrently with each other. However, when expanding a deterministic one to nondeterministic one that looks more natural, it is indispensable to consider invisible transition of each agent, which requires other information besides given communication channels. Therefore, non-deterministic controller synthesis has danger of decreasing concurrency of a synthesized program.

4 Related Works

We compare our verification and synthesis methods with related works.

Verification: One of common goals of other works [Cherkasova 87, Howell 88, Suzuki 89] is to uniformly specify most Petri net properties with temporal logic. Therefore its decision procedure inevitably falls into undecidable or costs more than the reachability problem. Complexity more than the reachability problem is out of the practical verification. Our method's complexity is almost equal to the coverability problem because of abandoning some verification properties, such as deadlock possibility. However, our method still provides more special properties, such as analysis of partial ordering among transition firing and mutual exclusion, which can not be covered by the traditional analysis. These abilities are effective for the concurrent program verification. Our method is not all-around but can complement the traditional analysis.

On the other hand, if we would restrict the Petri net into a bounded one, the verification becomes simpler [Uchihira 90a, Katai 82]. However, an unbounded buffer is sometimes necessary to specify ordinary concurrent programs. It might be possible to assume a large enough bounded buffer in real programs. A bounded Petri net with a large bounded buffer usually produces a larger coverability graph than ones of unbounded Petri nets. Recently, several efficient verification methods based on temporal logic model checking for bounded Petri nets have been proposed. They can be classified into two types; *symbolic model checking* [Hiraishi 95] and *partial order method* [Yoneda 93]. In particular, Yoneda et. al. proposed an efficient model checking method based on partial order for one-safe *time Petri nets* [Yoneda 93]. It would be a future promising approach to apply techniques of symbolic model checking and partial order method to unbounded Petri nets.

Synthesis: We think the software-reuse-based program synthesis is highly practical. Our method differs from other synthesis methods [Manna 84, Clarke 82] that also use temporal logic specifications, in regard to the point of utilizing software reuse. Another significant feature is to relax the automatic synthesis for only finite-state programs [Manna 84, Clarke 82] to infinite-state programs, such as a Petri net. Our method has the same approach as that of Valk and Jantzen [Valk 85] in point of tuning up existing programs (reusable software) satisfying the given specifications. However, our method has the following characteristic features:

- the specification is described with temporal logic, and
- the program synthesis method consists of two phases; controller synthesis and agent synthesis.

5 Summary

This chapter considers the fusion of unbounded Petri nets and temporal logic as a specification language for reactive and concurrent systems. We propose a version of the fusion and prove that the emptiness problem is decidable in this version. Then, verification and synthesis for reactive and concurrent systems are discussed based on these results. In this chapter,

(1) we define the class combining Petri nets and temporal logic which is decidable,

(2) the decision procedure for this class is applied to concurrent program verification, and

(3) a two phase synthesis method is provided which modifies reusable components to satisfy a specification.

This research was carried out to establish verification and synthesis for unbounded Petri nets and temporal logic. Efficient verification and synthesis for bounded Petri nets (i.e., transition systems) and temporal logic will be described in the subsequent chapters.

Appendix I

Results of Valk and Jantzen's method are briefly summarized. See the original paper [Valk 85] for proofs.

Definition 18

- $N = (P, T, w, m_0)$ *is a Petri net.*
- *A marking* m *is T'-continual for some subset T' of T, iff there is an infinite legal firing sequence from* m *which contains all* $t \in T'$ *infinitely often.*
- $CONTINUAL(T') \overset{def}{=} \{m \mid m$ *is T'-continual*$\}$.
- N *is a set of non negative integers. Let* $K \subset N^k$, *then the residue set of* K, *written* $res(K)$, *is a smallest subset of* K *which satisfies* $res(K) + N^k = K + N^k$.

Theorem 6

- $res(CONTINUAL(T'))$ *is finite and can be effectively constructed.*
- *Using* $res(CONTINUAL(T'))$, *we can construct a new Petri net* N' *whose all reachable markings are lying in* $CONTINUAL(T')$ *with the same number of places of* N, *but possibly additional transitions and arcs.*

Chapter 5

Compositional Verification Using Modal Logic

This chapter proposes PQL (Process Query Language) and the compositional verification method of reactive and concurrent systems using PQL. Our compositional approach is effective to ease the state explosion problem in the verification.

1 Background and Motivation

Temporal logic model-checking method [Clarke 86] is very useful for verification of reactive and concurrent systems. However, a major drawback to using this method is that as the scale of the programs increase, the computation costs for verification increase exponentially. An effective solution for this problem is compositional verification.

This chapter focuses on the compositional verification for finite state transition systems instead of Petri nets because of the following reasons.

- In general a compositional verification for infinite systems [Winskel 90] is restricted and difficult to apply to practical systems.
- Many practical reactive and concurrent systems can be modeled as finite state transition systems (with approximation).

Compositional verification for transition systems is formalized as process reduction in which the bisimulation equivalence of concurrent programs is used to extract from each system component (subprocess) only these abstract information necessary to verify each given query, thereby avoiding an explosion in cost.

In this chapter, PQL (Process Query Language) is proposed as an improved method in the solution of this problem. PQL is based on modal logic which is the union of temporal logic and process logic. Then, this chapter proposed the compositional verification method by using PQL with consideration of the divergence by internal transitions.We have applied this method to program verification of sequence control systems.

1.1 Background

Since the framework of reactive and concurrent systems often can be expressed with transition systems, verification methods based on temporal logic, process logic, CCS, ACP are well investigated [CAV 89, Beaten 90]. For example we have developed an automatic verification system (PTSV: Practical Temporal Specification and Verification tool) [Uchihira 89a] which is based on model-checking of CTL (Computation Tree Logic) [Clarke 86], and apply it to manufacturing systems.

The verification method based on model-checking can be summarized as follows. First, a flat and global finite-sate graph is generated, which expresses all possible behaviors of the target program. For example, when the target program consists of several processes, a flat and global finite-sate graph is generated by process composition based on interleaving semantics. Next, verification of whether given temporal logic queries are satisfied is performed by tracing all the states of the graph. However, this verification method has the following problems.

(1) **State explosion problem**
A generated global finite-state graph often becomes excessively large.

(2) **Lack of expression ability about "actions"**
CTL can express only queries about "states" and "state attributes".

One of the promising approaches for the problem (1) is *compositional verification* [Mishra 85, Clarke 89, Stirling 89b]. Compositional verification is defined as follows in this thesis [9] .

Compositional verification is a method for generating a local and minimum finite-state graph compositionally for each verification query and verifying it, instead of generating and verifying a global and huge graph.

Here, a local and minimum finite-state graph means a graph in which only necessary information to verify a given query is kept and the rest is reduced. Because this compositional verification method does not generate the global graph directly, the state explosion problem can be effectively eased.

Based on this idea, Mishra and Clarke applied the hierarchical verification method by CTL for verifying asynchronous circuits [Mishra 85]. However, this method has some limits for expressive power of verification queries and is not a sophisticated and general method. Also, Clarke, Long and MacMillan proposed a compositional verification method which introduced the compositional framework of CCS [Milner 89] (i.e., process composition and observation equivalence) for CTL model-checking. While CTL is a logic concerning "states" (state attributes), CCS is a calculus concerning "actions". Therefore, the point is how to handle "states" and "actions" in the same frame. This is also the solution for the problem (2). However, Clarke's verification method [Clarke 89] could not express "states" and "actions" freely mixed with each other. On the other hand, researches have been undertaken to characterize CCS expressions by *process logic* (ex. *Hennessy-Milner Logic* : *HML*) [Hennessy 85a, Hennessy 85b]. Furthermore, new logics, which combine temporal logic and process logic, have been investigated. These logics can express "states" and "actions" freely mixed with each other, and yet cope with compositional framework. They, therefore, are promising as query language for compositional verification. An example is Stirling's *General Temporal Logic* (*GTL*) [Stirling 89a, Stirling 89b]. GTL is a general logic which combines temporal logics (linear-time temporal logic and branching-time temporal logic) and process logic (HML). And further, Stirling and Walker proposed the compositional verification method by GTL [Stirling 89b].

1.2 Motivation

The point of compositional verification is to reduce (localize, minimize) the global state graph, leaving only essential information for each verification query. In the compositional verification by GTL, this information is formalized by *observation equivalence* [Milner 89] . Whenever the reduced graph is observation-equivalent to the original graph, it is assured that the reduction has no influence on the verification results, that is, it preserves the same results for every verification query.

However, the compositional verification method based on observation equivalence has one problem. In observation equivalence, any *divergence* is ignored. The *divergence* [Milner 81, Walker 90] means an infinite cycle (loop) of internal and unobservable transitions (i.e., τ-actions) which cannot be observed and then looks like deadlock for an observer (we call it an *external deadlock*). This occurs due to the fact that a certain fairness is assumed for internal and unobservable transitions (i.e., if there is at least one observable and executable action, there will not be an external deadlock by divergence). In the case of compositional verification, to make the transition observable or not will be decided by whether it is necessary for a given verification query or not. Therefore, if a difference in fairness exists depending upon the given verification query, a problem arises.

For instance, in the transition system T_a shown in Fig. 27, the answer will be NO for a query: "Does the system eventually reach the *end* state for all paths" because there exists a path with an infinite loop of b which never reaches the *end* state. On the other hand, the answer will be YES if b is unobservable. The reason is that T_a is recognized to be equivalent to T_b of Fig. 27 according to observation equivalence if b is unobservable (i.e., τ-action). However, it is a problem from the standpoint of compositional verification.

[9] The formal definition and survey of compositional verification are described in detail in Section 6.1.

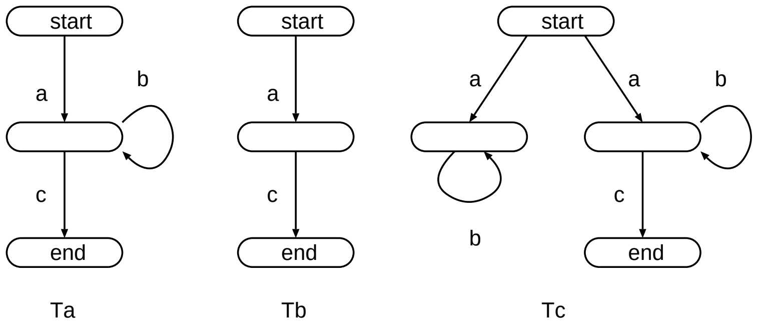


Figure 27. Transition systems with/without divergence

The existence of a path, which has an infinite loop of b and never reaches the *end* state, must be recognized regardless of observability of the action b.

In order to introduce expressive power of recognizing divergence into logics, Intuitionistic HML (IHML) [Stirling 87] has been proposed. Furthermore, Stirling suggested (but not with any concrete proposal) the possibility of adding the IHML feature to GTL and expanding it (GTL + IHML). But even if this "GTL + IHML" can be realized, there still remains the same problem. That is, the equivalence theory (equivalence based on partial bisimulation preorder) [Stirling 87], which IHML is based on, may sometimes produce recognition (abstraction from the observer's point of view) which does not agree with intuition, depending upon which action will be made observable.

In the case of T_a and T_c in Fig. 27, if b is not observable, divergence occurs in both transition systems, and T_a and T_c are equivalent from the equivalence theory of IHML. This equivalence is derived from the observation that both transition systems cannot take action c after action a *in some path.* However, a problem arises for the query: "Is there a state in which the system cannot take action c *in all paths* after action a." Intuitively, the answer should be NO for T_a and YES for T_c, regardless of whether b is observable or not. It is natural to be able to differentiate between these two cases. The equivalence theory of IHML, which considers both T_a and T_c to be equivalent, will cause a loss of the necessary information to verify the query. In other words, the compositional verification of GTL + IHML reduces T_c into T_a according to its equivalence theory, thereby creating stronger abstraction than necessary.

From these points of argument, it can be concluded that the usual equivalence theories (observation equivalence, equivalence by partial bisimulation preorder) which the usual modal logics (HML, GTL, GTL + IHML, etc.) based on, is aimed at equivalence relation from the viewpoint of the external processes as observers and therefore, it is not appropriate to apply directly these equivalence relations for compositional verification because of creating stronger abstraction than necessary. In compositional verification, "not observed" simply means "not paid attention" instead of "not synchronized". Therefore, even if a certain action is made unobservable, its existence must be preserved.

1.3 Overview of Main Results

In this chapter, we propose Process Query Language (PQL) for compositional verification in order to solve the problem described above. PQL can clearly express and easily handle the cycles of unobservable actions, taking a different approach from GTL + HML. The unique feature of PQL is the followings.

- PQL is be able to uniformly express both temporal logic properties and the existence of cycles of unobservable actions using the maximum/minimum fixed point operators.
- Constraints about state attributes and actions can be expressed in the uniformed and flexible way.
- PQL provides several useful macros including the regular expression.
- PQL model-checking is decidable and has the efficient decision procedure. Furthermore, PQL has a new process equivalence relation ($\pi\tau\omega$-bisimulation equivalence) which is used for the compositional verification. That is, if two transition systems (original one and reduced one) are equivalent, it is assured that every PQL query has the same answer.

1.4 Organization of the Chapter

The remainder of this chapter is organized as follows. Section 2 defines concurrent programs using transition systems and their equivalence relation (extended bisimulation equivalence). In Section 3, PQL is defined and it is shown that the discrimination ability of PQL is the same as the extended bisimulation equivalence. This means that even if the given concurrent program is reduced to a smaller equivalent program, verification results of PQL will be preserved. A compositional verification method using the result obtained in Section 3 is proposed and its effectiveness is demonstrated by means of experimental results in Section 4. In Section 5, we consider how to apply this method to actual reactive and concurrent systems. We introduce an application to chemical plant control systems by example. Section 6 mentions related works in which other compositional verification methods and a partial order method are surveyed and compared with our method.

2 Representation of Concurrent Programs

Concurrent programs are constructed from some number of processes. The program and each process are as transition systems $T = (S, P, A, \pi, \delta, s_0)$ [Arnold 92], which have been defined in Chapter 2.

Here, we assume that S, P, and A are finite sets for the automatic verification. In this case, the finite branching condition holds.

2.1 Equivalence of Transition Systems for Compositional Verification

We introduce a new bisimulation equivalence of transition systems (*$\pi\tau\omega$-bisimulation equivalence*), which is an extension of Milner's weak bisimulation equivalence [Park 81, Milner 89] in order to recognize τ-cycles. First, *(weak) bisimulation* used in CCS is defined for (S, Act, δ).

Definition 19 (bisimulation)
For (S, Act, δ), a binary relation $R \subset S \times S$ is (weak) bisimulation if $\forall(s,t) \in R$ *implies*

- $\forall a \in Act. \forall s' \in S.(if\ s \xrightarrow{a} s'\ then\ \exists t' \in S.t \overset{\hat{a}}{\Rightarrow} t' \wedge s'Rt')$
- $\forall a \in Act. \forall t' \in S.(if\ t \xrightarrow{a} t'\ then\ \exists s' \in S.s \overset{\hat{a}}{\Rightarrow} s' \wedge s'Rt')$

The τ-cycles cannot be recognized in *bisimulation*. Therefore, we introduce *$\tau\omega$-bisimulation* which can do.

Definition 20 ($\tau\omega$-divergence)
For (S, Act, δ) and $s \in S$,

$$s \uparrow \overset{def}{=} \forall n > 0. \exists s' \in S.s(\xrightarrow{\tau})^n s'$$

Definition 21 ($\tau\omega$-bisimulation)
For (S, Act, δ), a binary relation $R \subset S \times S$ is $\tau\omega$-bisimulation if $\forall(s,t) \in R$ *implies*

- $\forall a \in Act. \forall s' \in S.(if\ s \xrightarrow{a} s'\ then\ \exists t' \in S.t \overset{\hat{a}}{\Rightarrow} t' \wedge s'Rt')$
- $\forall a \in Act. \forall t' \in S.(if\ t \xrightarrow{a} t'\ then\ \exists s' \in S.s \overset{\hat{a}}{\Rightarrow} s' \wedge s'Rt')$
- $s \uparrow\ iff\ t \uparrow$

Furthermore, $\pi\tau\omega$-bisimulation is introduced for (S, Act, δ, P, π) where state attributes P and a boolean function π is added to (S, Act, δ) in order to take the equivalence of state attributes into consideration.

Definition 22 ($\pi\tau\omega$-bisimulation)
For (S, Act, δ), a binary relation $R \subset S \times S$ is $\pi\tau\omega$-bisimulation if $\forall(s,t) \in R$ *implies*

- $\pi(s) = \pi(t)$
- $\forall a \in Act. \forall s' \in S.(if\ s \xrightarrow{a} s'\ then\ \exists t' \in S.t \overset{\hat{a}}{\Rightarrow} t' \wedge s'Rt')$

- $\forall a \in Act. \forall t' \in S.$(*if* $t \xrightarrow{a} t'$ *then* $\exists s' \in S. s \overset{\hat{a}}{\Rightarrow} s' \wedge s'Rt'$)
- $s \uparrow$ *iff* $t \uparrow$

Definition 23 ($\approx, \approx_{\tau\omega}, \approx_{\pi\tau\omega}$)
In (S, Act, δ, P, π), *for* $s_1, s_2 \in S$, *if there exists bisimulation such that* $(s_1, s_2) \in R$, *then we denote* $s_1 \approx s_2$. $s_1 \approx_{\tau\omega} s_2$ *and* $s_1 \approx_{\pi\tau\omega} s_2$ *are defined in the same manner.*

Theorem 7 (Relation between $\approx, \approx_{\tau\omega}, \approx_{\pi\tau\omega}$**)**
For every $s_1, s_2 \in S$ *in* (S, Act, δ, P, π), *if* $s_1 \approx_{\pi\tau\omega} s_2$, *then* $s_1 \approx_{\tau\omega} s_2$, *and if* $s_1 \approx_{\tau\omega} s_2$, *then* $s_1 \approx s_2$.
Proof. This is clear from the definition. □

Since $s_1 \approx s_2$ is called *bisimulation equivalence*, $s_1 \approx_{\pi\tau\omega} s_2$ is to be called "*$\pi\tau\omega$-equivalence*".

Definition 24 ($\pi\tau\omega$-equivalence for Transition Systems)
For $T_1 = (S_1, P, A, \pi_1, \delta_1, s_{01})$ *and* $T_2 = (S_2, P, A, \pi_2, \delta_2, s_{02})$ ($S_1 \cap S_2 = \emptyset$ *is assumed*), T_1 *and* T_2 *is* $\pi\tau\omega$*-equivalence (denoted by* $T_1 \approx_{\pi\tau\omega} T_2$*) if* $s_{01} \approx_{\pi\tau\omega} s_{02}$ *in* $(S_1 \cup S_2, Act, \delta_1 \cup \delta_2, P, \pi_1 \cup \pi_2)$

$\pi\tau\omega$-equivalence is the extended bisimulation equivalence which can handles $\tau\omega$-divergence and state attributes, and it has higher discrimination ability than the bisimulation equivalence.

For divergence, there are several researches by Milner[Milner 81], Stirling[Stirling 87], Walker[Walker 90]. However, their equivalence ($T_1 \approx_p T_2 \overset{\text{def}}{=} T_1 \sqsubseteq T_2 \wedge T_2 \sqsubseteq T_1$) based on partial bisimulation preorder ($\sqsubseteq$) is weaker than $\tau\omega$-equivalence. For example (shown in Fig. 27), when b is unobservable (i.e. τ action), $T_a \approx T_b$ and $T_a \not\approx_{\tau\omega} T_b$, where τ-cycle can be discriminated. Also, as $T_a \approx_p T_c$ and $T_a \not\approx_{\tau\omega} T_c$, the problem pointed out in section 1 is solved.

Figure 28 shows the diagram of proper inclusions among several well-known equivalence relations and $\tau\omega$-bisimulation equivalence. The arrow ($\rightarrow$) means proper inclusion. For example, "$\tau\omega$-bisimulation equivalence $\rightarrow$ bisimulation equivalence" means "for every T_1,T_2 if $T_1 \approx_{\tau\omega} T_2$ then $T_1 \approx T_2$". The detail definitions of equivalence relations referred in this figure are shown in Appendix II.

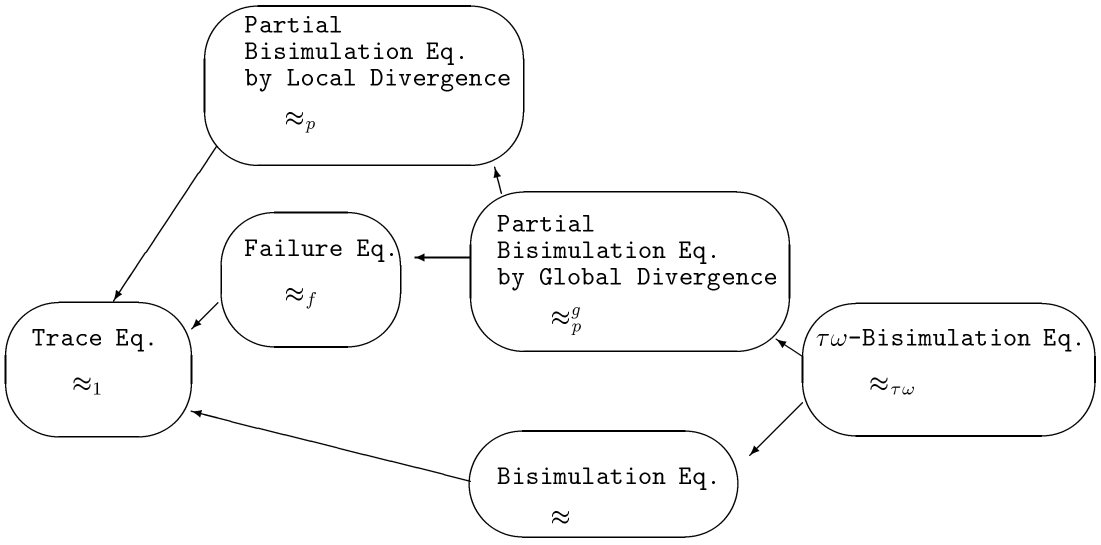

Partial Bisimulation Eq. by Local Divergence
$\approx_p$
Failure Eq.
$\approx_f$
Partial Bisimulation Eq. by Global Divergence
$\approx_p^g$
Trace Eq.
$\approx_1$
$\tau\omega$-Bisimulation Eq.
$\approx_{\tau\omega}$
Bisimulation Eq.
$\approx$

Figure 28. Comparison of equivalence relations

2.2 Composition of Transition Systems

A concurrent program is composed of processes (transition systems) which run concurrently and communicate with each other by the handshaking-type synchronization mechanism. Here, operators (composition,relabeling) concerning the composition of transition systems is to be introduced. Also, it will be proved that these operators preserve the $\pi\tau\omega$-equivalence.

Composition: $T_1 \mid T_2$

The transition system $T = T_1 \mid T_2$, which is composed of $T_1 = (S_1, P_1, A_1, \pi_1, \delta_1, s_{01})$ and $T_2 = (S_2, P_2, A_2, \pi_2, \delta_2, s_{02})$ such that $P_1 \cap P_2 = \emptyset$, is defined based on the interleaving semantics as follows:

$$T = (S_1 \times S_2, P_1 \cup P_2, A_1 \cup A_2, \pi, \delta, (s_{01}, s_{02}))$$

where $\pi : S_1 \times S_2 \to 2^{P_1 \cup P_2}$ is defined as
$\pi(s_1, s_2) = \pi_1(s_1) \cup \pi_2(s_2)$ for all $s_1 \in S_1, s_2 \in S_2$,
$\delta : S_1 \times S_2 \times (A_1 \cup A_2 \cup \{\tau\}) \to 2^{S_1 \times S_2}$ is defined as,

$$\delta((s_1, s_2), a) = \begin{cases} \{(s_1', s_2) \mid s_1' \in \delta_1(s_1, a)\} & \text{if } a \in A_1 \wedge a \notin A_2 \\ \{(s_1, s_2') \mid s_2' \in \delta_2(s_2, a)\} & \text{if } a \notin A_1 \wedge a \in A_2 \\ \{(s_1', s_2') \mid s_1' \in \delta_1(s_1, a), s_2' \in \delta_2(s_2, a)\} & \text{if } a \in A_1 \wedge a \in A_2 \\ \{(s_1', s_2') \mid (s_1' \in \delta_1(s_1, \tau), s_2' = s_2) \vee & \\ \qquad (s_2' \in \delta_2(s_2, \tau), s_1' = s_1)\} & \text{if } a = \tau \end{cases}$$

.

Intuitively, $T_1 \mid T_2$ means a concurrent program in which T_1 and T_2 run concurrently and take synchronous actions with the same labels, while $T_1 \mid T_2$ is formally defined as the above transition system T. In other words, $T_1 \mid T_2$ is equivalent to T based on the interleaving semantics.

relabeling: $T[f]$

For $T = (S, P, A, \pi, \delta, s_0)$ and a relabeling function $f = (f_A, f_P) such that f_A : A \cup \{\tau\} \to 2^{A' \cup \{\tau\}}$ (here, $f_A(\tau) = \{\tau\}$), and $f_P : P \to P' \cup \{true\}$ (here, $f_P(p) = true$ means it makes the state attribute p unobservable), the relabel-led transition system $T' = T[f]$ is defined as follows:

$$T' = (S, P', A', \pi', \delta', s_0)$$

where $\delta' : S \times (A' \cup \{\tau\}) \to 2^S$ is defined as $\delta'(s, a') = \{s' \mid s' \in \delta(s, a), a' \in f_A(a)\}$, and
$\pi' : S \to 2^{P' \cup \{true\}}$ is defined as $\pi'(s) = f_P(\pi(s))$.

Intuitively, $f_A(a) = \{a'\}$ and $f_P(p) = p'$ means simply relabeling a to a' and p to p', respectively. $f_A(a) = \{a_1, a_2\}$ means relabeling the original a to a_1 and its replica to a_2 after adding one more same transition with the label a.

Now, the actions and state attributes of the transition system T can be changed by the relabeling function f. This relabeling function is used to avoid overlapping of names of actions and state attributes in the case that the program consists of several same processes (i.e. $T' = T[f_1] \mid T[f_2]$). Also, the relabeling function is used to make actions and state attributes of T unobservable (i.e. $f_A(a) = \{\tau\}$, $f_P(p) = true$).

For convenience, this relabeling function f_A can be denoted as follows:

$$[\{l_{11}', ...l_{1k_1}'\}/l_1, ..., \{l_{n1}', ...l_{nk_n}'\}/l_n]$$

This denotes a function f_A such that $f_A(l_i) = \{l_{i1}', ...l_{ik_i}'\}$ for all $i \in \{1, ..., n\}$. This applies to f_P, too.

Theorem 8 (Preservation of $\pi\tau\omega$-equivalence)
If $T_{11} \approx_{\pi\tau\omega} T_{12}, T_{21} \approx_{\pi\tau\omega} T_{22}$, then $T_{11} \mid T_{21} \approx_{\pi\tau\omega} T_{12} \mid T_{22}, T_{11}[f] \approx_{\pi\tau\omega} T_{12}[f]$
Proof: *This is obvious for relabeling. As for composition, as mentioned in Proposition 7.2 of [Milner 89], it is easily shown that the following relation R is $\pi\tau\omega$-bisimulation:*
$R = \{((s_{11}, s_{21}), (s_{12}, s_{22})) \mid s_{11} \approx_{\pi\tau\omega} s_{12}, s_{21} \approx_{\pi\tau\omega} s_{22}, s_{11} \in S_{11}, s_{12} \in S_{12}, s_{21} \in S_{21}, s_{22} \in S_{22}\}\square$

3 Process Query Language

Process Query Language (PQL) is introduced which is used to describe queries in the compositional verification for transition systems.

The unique features of PQL is:

- Constraints about state attributes and actions can be expressed in the uniformed flexible way.
- Divergence by τ-cycles can be explicitly expressed using the maximum/minimum fixed point operators.
- Strong expression ability including the regular expression.

First, a modal logic SPQL (Strong Process Query Logic) is defined, where τ actions are observable, and then PQL is defined as a macro-language of SPQL where τ actions are unobservable.

3.1 SPQL (Strong Process Query Logic)

SPQL is a modal logic which unifies temporal logic and process logic with fixed point operators.

Definition 25 (SPQL formula)

[Syntax]
P : *a set of state attributes*
A : *a set of actions*

SPQL formulas are recursively defined as follows. Here, a free state variable is a state logical variable which is not bound by any fixed point operator (μ-operator).

State Formula
- *A state logical variable Z is a state formula.*
- *$p \in P$ and true are state formulas.*
- *If f_1 and f_2 are state formulas, then $f_1 \wedge f_2$ and $\neg f_1$ are state formulas.*
- *If f is a state formula, and Z is a free state logical variable appearing in f, and negation nesting of Z in f is even, then $\mu Z.f$ is a state formula.*
- *If g is a path formula, $\exists g$ is a state formula.*

Path Formula
- *$a \in A$ is a path formula.*
- *If g_1 and g_2 are path formulas, then $g_1 \wedge g_2$ and $\neg g_1$ are path formulas.*
- *If f is a state formula, then Xf and Tf are path formula.*

SPQL Formula
- *If a state formula f include no free state logical variables, then f is a SPQL formula.*

[Semantics]

A state satisfying an SPQL formula f is called a model of f. $V[\![f]\!]$, a set of models of SPQL formula f for a transition system $T = (S, P, A, \pi, \delta, s_0)$, will be defined. Here, we assume the following notations.

- f : *a state formula.*
- g : *a path formula.*
- SF : *a set of state formulas.*
- PF : *a set of path formulas.*
- $V[\![f]\!] \subset S$: *a set of states satisfying f.*
- $R[\![g]\!] \subset S \times S$: *a set of paths, whose length is 1 (i.e., edges), satisfying g.*
- $(\lambda Z.f_1)f_2$: *a state formula in which a free state logical variable Z in f_1 is replaced with f_2.*
- $[\![S']\!]^{-1}$: *a virtual state formula f such that $V[\![f]\!] = S'$ (for example, $[\![\emptyset]\!]^{-1} = false, [\![S]\!]^{-1} = true$).*

$V : SF \to 2^S$ for a state formula f, which includes no free state logical variables, is defined as follows:

$V[\![p]\!] = \{s \in S \mid p \in \pi(s)\}$
$V[\![\exists g]\!] = \{s \in S \mid \exists (s, s') \in R[\![g]\!])\}$
$V[\![\neg f]\!] = S - V[\![f]\!]$
$V[\![f_1 \wedge f_2]\!] = V[\![f_1]\!] \cap V[\![f_2]\!]$
$V[\![\mu Z.f]\!] = \bigcap \{S' \subseteq S \mid V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!] \subseteq S'\}$

$R : PF \to 2^{S \times S}$ is defined as follows:

$R[\![a]\!] = \{(s, s') | s \xrightarrow{a} s'\}$
$R[\![\neg g]\!] = (S \times S) - R[\![g]\!]$
$R[\![g1 \wedge g2]\!] = R[\![g_1]\!] \cap R[\![g_2]\!]$
$R[\![Xf]\!] =$
$\{(s, s') | \exists a.(s \xrightarrow{a} s' \wedge a \neq \tau \wedge s' \in V[\![f]\!])\}$
$R[\![Tf]\!] =$
$\{(s, s') | \exists a.(s \xrightarrow{a} s' \wedge a = \tau \wedge s' \in V[\![f]\!])\}$

Also, the following convenient constants and operators are introduced.

- $false \overset{\text{def}}{=} \neg true$
- $f_1 \vee f_2 \overset{\text{def}}{=} \neg(\neg f_1 \wedge \neg f_2)$
- $\nu Z.f \overset{\text{def}}{=} \neg \mu Z'.\neg(\lambda Z.f)(\neg Z')$

The intuitive meaning of each operator is shown as follows.

- $\wedge$(AND),$\vee$(OR),$\neg$(NOT).
- Xf : f will be true immediately after any action except τ occurs.
- Tf : f will be true immediately after a τ action occurs.
- $\exists g$: g is true on some path.
- $\mu Z.f$: μ is the minimum fixed point operator.

  $\mu Z.f$: expresses the minimum fixed point where a free state logical variable Z in f is recursively bound with f itself.

  For instance, $\mu Z.(f \vee \exists XZ)$ means “on some paths, f will be eventually satisfied”.
- $\nu Z.f$: ν is the maximum fixed point operator.

  For instance, $\nu Z.(f \vee \exists XZ)$ means “on some paths, f will be eventually satisfied, or there exists an infinite path (it cannot be decided by a finite path)”.

The important properties of the minimum and maximum fixed point operators of SPQL are stated in the following lemmas. Here, it is defined that for $\lambda x.y$, $(\lambda x.y)^1 z \overset{def}{=} (\lambda x.y)z$ and $(\lambda x.y)^{k+1} z \overset{def}{=} (\lambda x.y)(\lambda x.y)^k z$ $(k > 1)$.

Lemma 5 (Monotonicity)

$$S_1 \subset S_2 \;\Rightarrow\; V[\![(\lambda Z.f)[\![S_1]\!]^{-1}]\!] \subset V[\![(\lambda Z.f)[\![S_2]\!]^{-1}]\!]$$

Proof:
As the number of the negation nesting on Z in f is even, the monotonic property is clear by the definition of SPQL. $\Box$

Lemma 6 (Properties of minimum/maximum fixed point operators)
(1)

$$\begin{aligned} V[\![\mu Z.f]\!] &= \lim_{k\to\infty} (\lambda S'.V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k \emptyset \\ &= \lim_{k\to\infty} V[\![(\lambda Z.f)^k false]\!] \\ V[\![\nu Z.f]\!] &= \lim_{k\to\infty} (\lambda S'.V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k S \\ &= \lim_{k\to\infty} V[\![(\lambda Z.f)^k true]\!] \end{aligned}$$

(2) $$s \in V[\![\mu Z.f]\!] \iff \exists k.\forall h \geq k.s \in V[\![(\lambda Z.f)^h false]\!]$$

(3) If S is finite, for any f including μ operators, there exists a formula f^* of finite length f^* including no μ operators such that $V[\![f]\!] = V[\![f^*]\!]$.
Proof:

(1) *Let $S^k = (\lambda S'.V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k \emptyset = V[\![(\lambda Z.f)^k false]\!]$ and $S^\omega = \lim_{k\to\infty} S^k$. According to the monotonicity,*

$$\forall S' \subset S.(V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!] \subset S' \ \Rightarrow\ S^\omega \subset S').$$

Also, it is clear that

$$V[\![(\lambda Z.f)[\![S^\omega]\!]^{-1}]\!] \subset S^\omega,$$

therefore,

$$V[\![\mu Z.f]\!] = \bigcap\{S' \subseteq S \mid V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!] \subseteq S'\} = S^\omega.$$

Same for $V[\![\nu Z.f]\!]$.

(2) *Since $\emptyset \subset S^1 \subset S^2 \cdots \subset S^\omega$,*

$$s \in S^\omega \iff \exists k.\forall h \geq k.s \in S^h \iff \exists k.\forall h \geq k.s \in V[\![(\lambda Z.f)^h false]\!].$$

(3) *When S is finite, from (2)*

$$\exists k.(V[\![\mu Z.f]\!] = V[\![(\lambda Z.f)^k false]\!]).$$

Therefore, there exists a formula f', in which the most outer μ-type subformula $\mu Z.f_{sub}$ of f is substituted with $(\lambda Z.f_{sub})^k false$, such that $V[\![f]\!] = V[\![f']\!]$. By repeating this substitution operation until there is not any more μ-type subformula, we can construct a formula f^* including no μ operators such that $V[\![f]\!] = V[\![f^*]\!]$.

□

Definition 26 (Model)
If $s \in S$ and $s \in V[\![f]\!]$, s is called a model of an SPQL formula f, and is denoted as $s \models f$. Similarly, if $s_0 \in V[\![f]\!]$ for a transition system $T = (S, P, A, \pi, \delta, s_0)$, T is a model of an SPQL formula f, and denoted as $T \models f$.

Theorem 9 (Decidability of SPQL model checking)
For any transition system $T = (S, P, A, \pi, \delta, s_0)$ and any SPQL formula f, if S is finite, there exists an algorithm which automatically decides whether $T \models f$.
Proof:
It is enough to show the algorithm which computes $V[\![\mu Z.f]\!]$ and $V[\![\nu Z.f]\!]$. From lemma 6(1),

$$[\![\mu Z.f]\!] = \lim_{k\to\infty} (\lambda S'.V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k \emptyset.$$

Therefore, if S is finite, it is decidable since $(\lambda S'.V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k \emptyset$ converges in a finite number k. Same for $V[\![\nu Z.f]\!]$. □

The efficient model checking algorithm for SPQL can be implemented by extending those for CTL [Clarke 86].

3.2 PQL (Process Query Language)

PQL is defined based on SPQL. In SPQL, the τ action is assumed to be observable and it has to be explicitly expressed in an SPQL formula like Tf. PQL is a language in which the τ action is unobservable, and therefore it is not necessary to express the τ action explicitly in the formulas.

Definition 27 (PQL formula)
[Syntax]
P : *a set of state attributes.*
A : *a set of actions.*

- *A state logical variable* Z *is a state formula.*
- $p \in P$ *and* $true$ *are state formulas.*
- *If* f_1 *and* f_2 *are state formulas, then* $f_1 \wedge f_2$ *and* $\neg f_1$ *are state formulas.*
- *If* f *is a state formula and* $a \in A$, $\ll a \gg^+ f$, $\ll a \gg^- f$, $\ll . \gg^+ f$, $\ll . \gg^- f$, $\ll\gg^+ f$, $\ll\gg^- f$ *are state formulas.*
- *If* f *is a state formula, and* Z *is a free state logical variable appearing in* f*, and negation nesting of* Z *in* f *is even, then* $\mu Z.f$ *is a state formula.*
- *If a state formula* f *include no free state logical variables, then* f *is a PQL formula.*

The set of entire PQL formulas is denoted as L_{PQL}.
[Semantics]
The PQL formula can be translated into an equivalent SPQL formula with the following rules. Therefore the semantics of a PQL formula is given as one of the translated SPQL formula.

- $\ll a \gg^- f \Leftrightarrow$
  $\mu Z_1.(\exists((a \wedge X(\mu Z_2.(f \vee \exists T Z_2))) \vee T Z_1))$
- $\ll a \gg^+ f \Leftrightarrow$
  $\nu Z_1.(\exists((a \wedge X(\nu Z_2.(f \vee \exists T Z_2))) \vee T Z_1))$
- $\ll . \gg^- f \Leftrightarrow$
  $\mu Z_1.(\exists(X(\mu Z_2.(f \vee \exists T Z_2)) \vee T Z_1))$
- $\ll . \gg^+ f \Leftrightarrow$
  $\nu Z_1.(\exists(X(\nu Z_2.(f \vee \exists T Z_2)) \vee T Z_1))$
- $\ll\gg^- f \Leftrightarrow \mu Z.(f \vee \exists T Z)$
- $\ll\gg^+ f \Leftrightarrow \nu Z.(f \vee \exists T Z)$
- *Other PQL formulas are also SPQL formulas.*

The intuitive meaning of each operator is shown as follows.

- $\ll a \gg^- f$: After an action a, it is possible to become a state in which f is satisfied.
- $\ll a \gg^+ f$: After an action a, it is possible to become a state in which f is satisfied, or there is a divergence of the τ-cycle.

When there exists a divergence of the τ-cycle and it is impossible to become a state in which f is satisfied after an action a, $\ll a \gg^- f$ is interpreted as $false$ and $\ll a \gg^+ f$ is interpreted as $true$. In the latter, the formula interpreted as true because the system never reaches a state in which f is *unsatisfied* when it falls into the divergence.

For other PQL formulas, $\ll . \gg f$ means $\exists a \in A. \ll a \gg f$, $\ll\gg f$ means $\ll \varepsilon \gg f$.

PQL has a stronger expression ability than one of CTL, that is, PQL can express anything which CTL can. PQL can express various kinds of verification queries about actions and state attributes of concurrent programs from the temporal point of view. However, it cannot be said that PQL's readability is excellent enough for practical use, so the following macro-operators are introduced for convenience. Here, *apat*, *apet*, *epat*, and *epet* means $\forall\Box$, $\forall\Diamond$, $\exists\Box$, and $\exists\Diamond$ of CTL temporal operators, respectively. That is, $a =$ "$\forall$(for all)", $p =$"path", $e =$ "$\exists$ (exists)" , and $t =$ "time".

- $f_1 \vee f_2 \overset{\mathrm{def}}{=} \neg(\neg f_1 \wedge \neg f_2)$
- $(\lambda Z.f_1)f_2$ (λ notation is allowed to be appear explicitly in PQL formulas)
- $\nu Z.f \overset{\mathrm{def}}{=} \neg\mu Z'.\neg(\lambda Z.f)(\neg Z')$
- $[[a]]^+ f \overset{\mathrm{def}}{=} \neg \ll a \gg^- \neg f$
- $[[a]]^- f \overset{\mathrm{def}}{=} \neg \ll a \gg^+ \neg f$
- $[[.]]^+ f \overset{\mathrm{def}}{=} \neg \ll . \gg^- \neg f$
- $[[.]]^- f \overset{\mathrm{def}}{=} \neg \ll . \gg^+ \neg f$
- $[[\ ]]^+ f \overset{\mathrm{def}}{=} \neg \ll\gg^- \neg f$
- $[[\ ]]^- f \overset{\mathrm{def}}{=} \neg \ll\gg^+ \neg f$
- $apat\ f \overset{def}{=} [[]]^+ \nu Z.(f \wedge [[.]]^+ Z)$
  (f can always be true in all paths)
- $apet\ f \overset{def}{=} [[]]^+ \mu Z.(f \vee (\ll . \gg^-\ true \wedge [[.]]^+ Z))$
  (f can eventually be true in all paths)
- $epat\ f \overset{def}{=} \ll\ \gg^-\ \nu Z.(f \wedge ([[.]]^+ false \vee \ll . \gg^-\ Z))$
  (f can always be true in some paths)
- $epet\ f \overset{def}{=} \ll\ \gg^-\ \mu Z.(f \vee \ll . \gg^-\ Z)$
  (f can eventually be true in some paths)
- $external\_deadlock \overset{def}{=} [[.]]^+ false$
  (It looks deadlock from the observer, although the divergence may occur.)
- $internal\_divergence \overset{def}{=} \ll\ \gg^+\ false$
  (The divergence exists which cannot be observed.)
- $internal\_deadlock \overset{def}{=} [[.]]^+ false \wedge \neg \ll\ \gg^+\ false$
  (It is complete deadlock without divergence)

We show several examples of PQL formulas. We remark that PQL can express temporal properties equivalent to th regular expression.

Example 7 *Examples of verification queries by PQL*

(1) $apat(epet \ll a \gg^-\ true)$

Meaning: an action a *is deadlock-free.*

(2) $\nu Z_1.([[b]]^+ false \wedge [[c]]^+ false \wedge [[a]]^+(\nu Z_2.([[a]]^+ false \wedge [[b]]^+ Z_2 \wedge [[c]]^+ Z_1)))$

Meaning: The sequence of actions a,b, and c are equivalent to th regular expression $(ab^*c)^*$*).*

As one of the most important properties of PQL, it is proved that the discrimination ability of PQL for transition systems is equal to that of $\pi\tau\omega$-equivalence. That is, the transition systems which are $\pi\tau\omega$-equivalent have the same results for any PQL verification queries. This property is necessary for the compositional verification.

Definition 28 ($\approx_{PQL}$)

$$T_1 \approx_{PQL} T_2 \overset{def}{=} \forall f \in L_{PQL}.(T_1 \models f \text{ iff } T_2 \models f)$$

Theorem 10 (Relationship between PQL and $\pi\tau\omega$-equivalence)
For any transition systems T_1 and T_2,

$$T_1 \approx_{\pi\tau\omega} T_2 \Leftrightarrow T_1 \approx_{PQL} T_2$$

Proof: Appendix □

Example 8 *Simple verification example by PQL*

For three transition systems in Figure 27, when an action b is assumed to be unobservable (i.e. tau action), verification (model checking) results for three PQL formulas are shown in th Table 5. In this table, YES means the transition system is a model of the PQL formula, and NO means it is not. These results show that there exist PQL formulas which can discriminate these three transition systems. It supports that $T_a \not\approx_{\pi\tau\omega} T_b$, $T_a \not\approx_{\pi\tau\omega} T_c$, and $T_b \not\approx_{\pi\tau\omega} T_c$.

Table 5. Simple Example of Verification Using PQL

PQL Formula	T_a	T_b	T_c
$\ll a \gg^- \ll c \gg^- true$	YES	YES	YES
$[[a]]^- \ll c \gg^- true$	NO	YES	NO
$\ll a \gg^- \neg \ll c \gg^- true$	NO	NO	YES

4 Compositional Verification

The compositional verification method for concurrent programs, based on the result of Theorem 10, is shown here. First, the *verification scope* is introduced which is necessary for compositional verification, then the verification procedure is proposed.

4.1 Verification Scope

Generally, each of verification queries often may concern only a local properties of the target systems. Therefore, we introduce a verification scope in order to explicitly state what potion of the program is exclusively verified.

To be more specific, the verification scope VS of a transition system $T = (S, P, A, \pi, \delta, s_0)$ is expressed as $VS = (P', A')$, where $P' \subset P$ and $A' \subset A$ are watched state attributes and actions in the verification. Then a verification query is expressed as a pair of a verification scope VS and a PQL formula f, (VS, f). We remark that state attributes P_f and actions A_f appearing in f must be included in the verification scope $VS = (A', P')$ (i.e., $P_f \subset P', A_f \subset A'$). In the verification procedure, all the state attributes and actions which are not on the verification scope VS are considered unobservable, and relabeled with τ and *true*. This relabeling function derived from VS is denoted as l_{VS}.

Accordingly, $T \models (VS, f)$, a macro-expression for describing verification queries easily, is defined as $T[l_{VS}] \models f$. It is more natural for verifier to describe $T \models (VS, f)$ in which VS is part of the verification query instead of $T[l_{VS}] \models f$. For instance, for a query $f = \ll . \gg^- true$, there may be some observable actions which do not appear in f explicitly, which can be expressed as observable using a verification scope.

4.2 Verification Procedure

A basic idea of the compositional verification is to achieve the global verification as the composition of local verifications. In this case, instead of verifying globally

$$T \models (VS_1, f_1) \wedge (VS_2, f_2) \wedge ... \wedge (VS_n, f_n),$$

we verify compositionally

$$T_1 \models f_1 \wedge T_2 \models f_2 \wedge ... \wedge T_n \models f_n.$$

Here, T_i is the projection of T onto the verification scope VS_i of f_i. This means that state attributes and actions of T which are not appeared in VS_i are interpreted as unobservable and T is reduced to T_i as much as possible such that $T[l_{VS_i}] \approx_{PQL} T_i$ using the reduction function described later. T_i is called the projection transition system of T to VS_i. The more local the scope VS_i is, the smaller the projection transition system T_i is. These smaller projection transition systems can ease the state explosion problem which the naive construction of T often causes.

It is shown how to construct the projection transition system as follows (Fig. 29). We only show the case that the concurrent program $T = T_1 \mid T_2$ consists of two processes (transition systems), T_1 and T_2. When the program consists of n processes, it can be treated as an extension of one of 2 processes (i.e., $T = (...((T_1 \mid T_2) \mid T_3)... \mid T_n))$. The verification query is expressed as (VS, f).

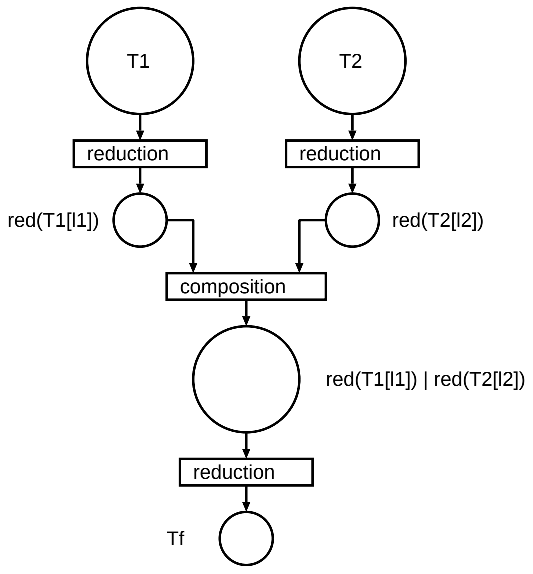


Figure 29. Construction of projection transition systems

If T is composed directly from T_1 and T_2, the size of T is of the multiplication order of T_1 and T_2, which is the cause of its state explosion. The following procedure to compose the projection transition system T_f can ease the state explosion problem using a reduction function.

$$T_f = red((red(T_1[l_1]) \mid red(T_2[l_2]))[l_{VS}])$$

Here, l_1 and l_2 are relabeling functions that make attributes and actions unobservable which do not appear in the verification scope of f and do not related to the composition (i.e., synchronization) of T_1 and T_2. That is, these attributes and actions are relabeled as *true* and τ. Also, *red* is a reduction function and $T' = red(T)$ is a reduced transition system such that $T \approx_{\pi\tau\omega} T'$ and $\mid T \mid \geq \mid T' \mid$. The concrete reduction function will be mentioned in the following section.

Based on Theorem 9, the model checking can be done for the projection transition system T_f instead of $T[l_{VS}]$. As $T[l_{VS}] \approx_{\pi\tau\omega} T_f$, it is guaranteed that they have the same verification result (i.e., $T[l_{VS}] \approx_{PQL} T_f$) by Theorem 10. Since this compositional verification method can control the maximum number of states of the temporary transition systems which are created during repeating composition and reduction, it is possible to verify large-scale concurrent programs.

The following is the input and output of an automatic verification tool using this method (Fig. 4.2) :

INPUT:

(1) Concurrent Program
(expressed in terms of a composition of transition systems, including necessary relabeling functions).

(2) Verification Queries
(each query is expressed as a pair of verification scope and PQL formula)

OUTPUT:

(1) Answers to Each Verification Queries
(YES/NO)

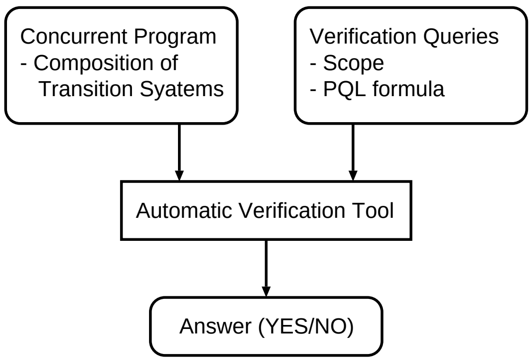


Figure 30. Input and output for the automatic verification tool

From above input information, the verification tool can perform automatically relabeling, reduction, composition, and model checking, then output answers.

4.3 Reduction Function

Reduction means generating the smaller transition system $T' = red(T)$ from a given transition system T such that T' is $\pi\tau\omega$-equivalent to T. Two reduction functions are introduced here.

(1) Reduction by $\pi\tau\omega$-bisimulation

Let R be the maximum $\pi\tau\omega$-bisimulation in T. The transition system, where all related (i.e., equivalent) nodes in R are reduced into one node, is called $red(T)$. Then $T' = red(T)$ is the transition system with minimum states which satisfies $T' \approx_{\pi\tau\omega} T$. The efficient algorithm to compute $\pi\tau\omega$-bisimulation R is acquired by modification of an algorithm for bisimulation [Kanellakis 90].

(2) Reduction by rewriting rules

Although the reduction by $\pi\tau\omega$-bisimulation can obtain the the transition system with minimum states, its computation cost might swell in case when T is large. So, five heuristic rules (shown in Fig. 4.3) that rewrite a transition system focused on the *tau* actions are applied to reduce the transition system as much as possible. Since rewriting by these heuristic rules preserves the $\pi\tau\omega$-equivalence, the reduced transition system $T' = red(T)$ satisfies $T' = red(T)$ は $T' \approx_{\pi\tau\omega} T$. If compared with original reduction by $\pi\tau\omega$-bisimulation, the reduction rate may be smaller, but it possesses the capability of speedy reduction for large scale transition systems. Furthermore, the combination of two reduction methods is effective in which heuristic rewriting rules are applied first before reduction by $\pi\tau\omega$-bisimulation.

4.4 Experiments

4.4.1 The Jobshop

Using the simple example presented in "Communication and Concurrency" by Robin Milner [Milner 89], we confirm effectiveness of the compositional verification. The target program, *jobshop*, which we want to verify, is given as the following composition of four process ($jobber \times 2, hammer, mallet$).

$$jobshop = (((jobber[l_{j1}] \mid hammer[l_h]) \mid mallet[l_m]) \mid jobber[l_{j2}])[l_{js}]$$

The transition systems of these processes are shown in Fig. 32, and labeling functions are defined as follows.

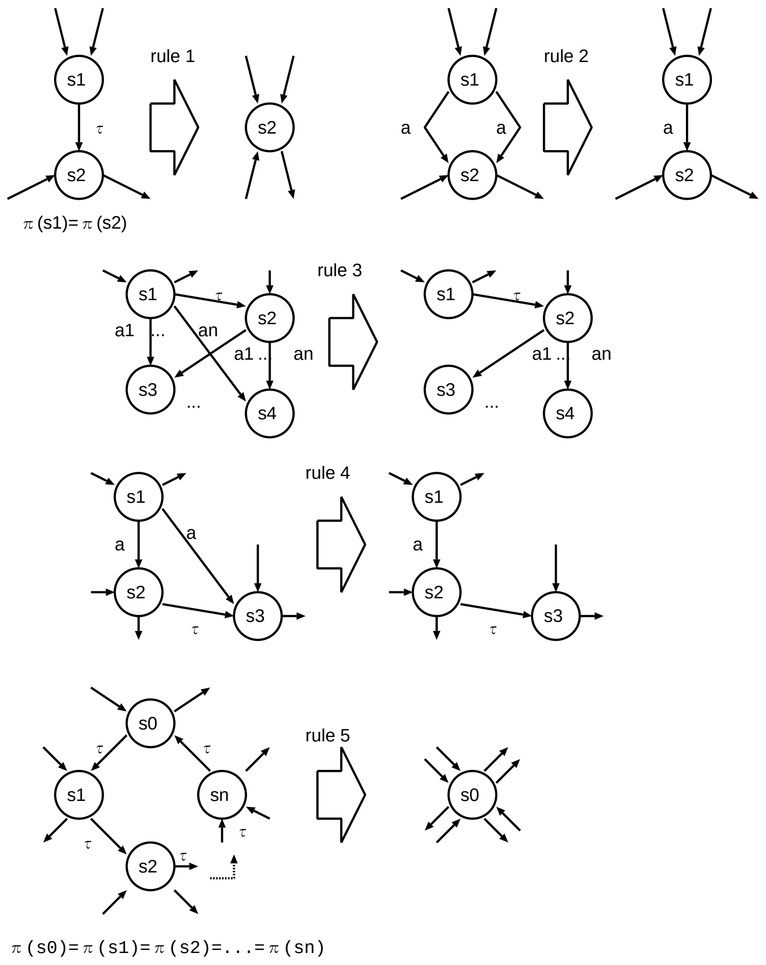


Figure 31. Heuristic reduction rules

- $l_{j1} = ([\tau/easy, \tau/hard, \tau/normal, \tau/do, in1/in, out1/out,$ $geth1/geth, puth1/puth, getm1/getm, putm1/putm], [\,])$.
- $l_{j2} = ([\tau/easy, \tau/hard, \tau/normal, \tau/do, in2/in, out2/out,$ $geth2/geth, puth2/puth, getm2/getm, putm2/putm], [\,])$.
- $l_h = ([\tau/error, \{geth1, geth2\}/geth, \{puth1, puth2\}/puth], [h\_off/off])$.
- $l_m = ([\tau/error, \{getm1, getm2\}/getm, \{putm1, putm2\}/putm], [m\_off/off])$.
- $l_{js} = ([\tau/geth1, \tau/geth2, \tau/puth1, \tau/puth2, \tau/getm1,$ $\tau/getm2, \tau/putm1, \tau/putm2, in/in1, in/in2, out/out1, out/out2], [])$.

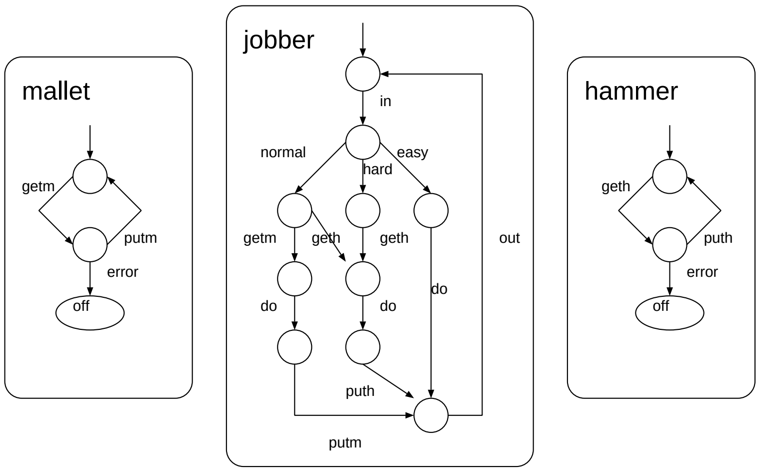


Figure 32. Example: The Jobshop

The following verification queries are given to the target program.

(1) Can the deadlock occur? $(([\,], [\,]), epet\ external\_deadlock)$

(2) Is there any internal divergence which cannot be observed? $(([\,], [\,]), epet\ internal\_divergence)$

(3) Is there a pattern of the action sequence "in, in, out, out"?
$(([in, out], [\,]), \ll in \gg^{-} \ll in \gg^{-} \ll out \gg^{-} \ll out \gg^{-} true)$

(4) Even if *mallet* is not available, if *hammer* is available, deadlock will never occur. Is it true?
$(([\,], [m\_off, h\_off]), apat((m\_off \wedge \neg \ll\ \gg^{-} h\_off) \supset \neg external\_deadlock))$

How the verification system works for above queries is shown in the following two steps.

(Step1) Composition and Reduction: The projection transition system is composed for each query in which unrelated actions and state attributes for the query are reduced as much as possible. For instance, for query (3), the projection transition system ($jobshop_{f_3}$) is composed in the following way. We remark that this composition procedure is automatically made of the original composition structure of *jobshop* and the verification scope.

$$\begin{aligned} jobshop_{f_3} = {} & red(red((red((red((red(jobber[l_{j1}]) \mid red(hammer[l_{h'}]))[l_{j1h}]) \\ & \mid red(mallet[l_{m'}]))[l_{j1hm}]) \mid red(jobber[l_{j2}]))[l_{j1hmj2}])[l_{js'}]), \end{aligned}$$

where

- $l_{h'} = ([\tau/error, \{geth1, geth2\}/geth, \{puth1, puth2\}/puth], [true/off])$.

- $l_{m'} = ([\tau/error, \{getm1, getm2\}/getm, \{putm1, putm2\}/putm], [true/off])$.
- $l_{j1h} = [\tau/geth1, \tau/puth1]$
- $l_{j1hm} = [\tau/getm1, \tau/putm1]$
- $l_{j1hmj2} = [\tau/geth2, \tau/puth2, \tau/getm2, \tau/putm2]$
- $l_{js'} = [in/in1, in/in2, out/out1, out/out2]$
- l_{j1} and l_{j2} are not changed.

Table 6 shows the final state numbers of *jobshop* (original one) and $jobshop_{f_3}$ (reduced one), and maximum numbers of states in two cases which are temporally created during composition and reduction procedure. The two previously mentioned reduction functions are used here. This table shows that our reduction method could reduce the final size of states to one-tenth of the original one.

Table 6. Effect of Reduction (The Jobshop)

Transition System	*Final Size*	*Maximum Temporary Size*
jobshop (original)	164	164
$jobshop_{f_3}$ (reduction function (1))	17	72
$jobshop_{f_3}$ (reduction function (2))	90	110

(Step2) Verification: For each query f and its projection transition system $jobshop_f$, the decision (YES/NO) on $jobshop_f \models f$ is made. The results of these decisions are shown in Table 7.

Table 7. Result of Verification (The Jobshop)

Verification Query	*Result*
(1)	YES
(2)	YES
(3)	YES
(4)	YES

4.4.2 The Manufacturing Machine Control Software

Our compositional verification method was applied to a middle-scale manufacturing machine control software (Fig. 33). This machine consists of 5 arms, 4 chambers, and other equipment. An etching chamber, a transfer chamber, an electrode, a pusher, and 2 inner arms constitute an etching unit. Two etching units are identical. The outer arm repeatedly transports material wafers from a loading cassette to one of two pushers in front of chambers (vacant one) via a wafer liner and from the pusher to an unloading cassette. The wafer liner puts a wafer in order. One inner arm transports material wafers from a pusher to an electrode in an etching chamber, and the other inner arm does in the opposite directory, that is, from an electrode to a pusher. In etching chamber, materials should be etched in a vacuum. The two gates are controlled to keep the etching chamber in a vacuum. Since these 5 arms can move concurrently, the control software becomes considerably complicated.

This machine control software is controlled by a concurrent (multi-task) program which consists of 16 element processes (tasks). Each process is a local controller of a corresponding controlled object (i.e., arm, chamber, electrode, etc.). This control software is modeled by a set of communicating transition systems shown in Fig. 34 where transition systems are represented by safe Petri nets and synchronization between transitions is represented by a dotted line. Table 8 shows the sizes of element processes. The state numbers of each element process may seem to be small. It attributes to the fact that only synchronization parts of systems are modeled by transition systems.

The following verification queries are given for the software.

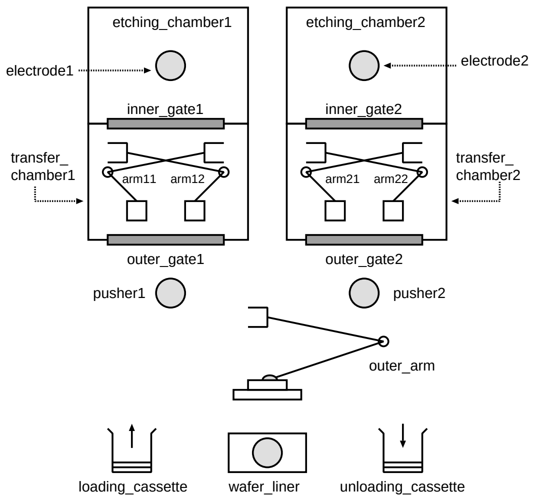


Figure 33. Manufacturing Machine

Table 8. Machine Control Software

Element Process × The number of it	*State Number of Transition System*
(p1) Outer Arm × 1	6
(p2) Wafer Liner × 1	3
(p3) Loading Cassette × 1	5
(p4) Electrode × 2	3
(p5) Unloading Cassette × 1	5
(p6) Setting Arm (arm_{i1}) × 2	6
(p7) Extracting Arm (arm_{i2}) × 2	6
(p8) Transfer Chamber × 2	2
(p9) Etching Chamber × 2	4
(p10)Pusher × 2	3

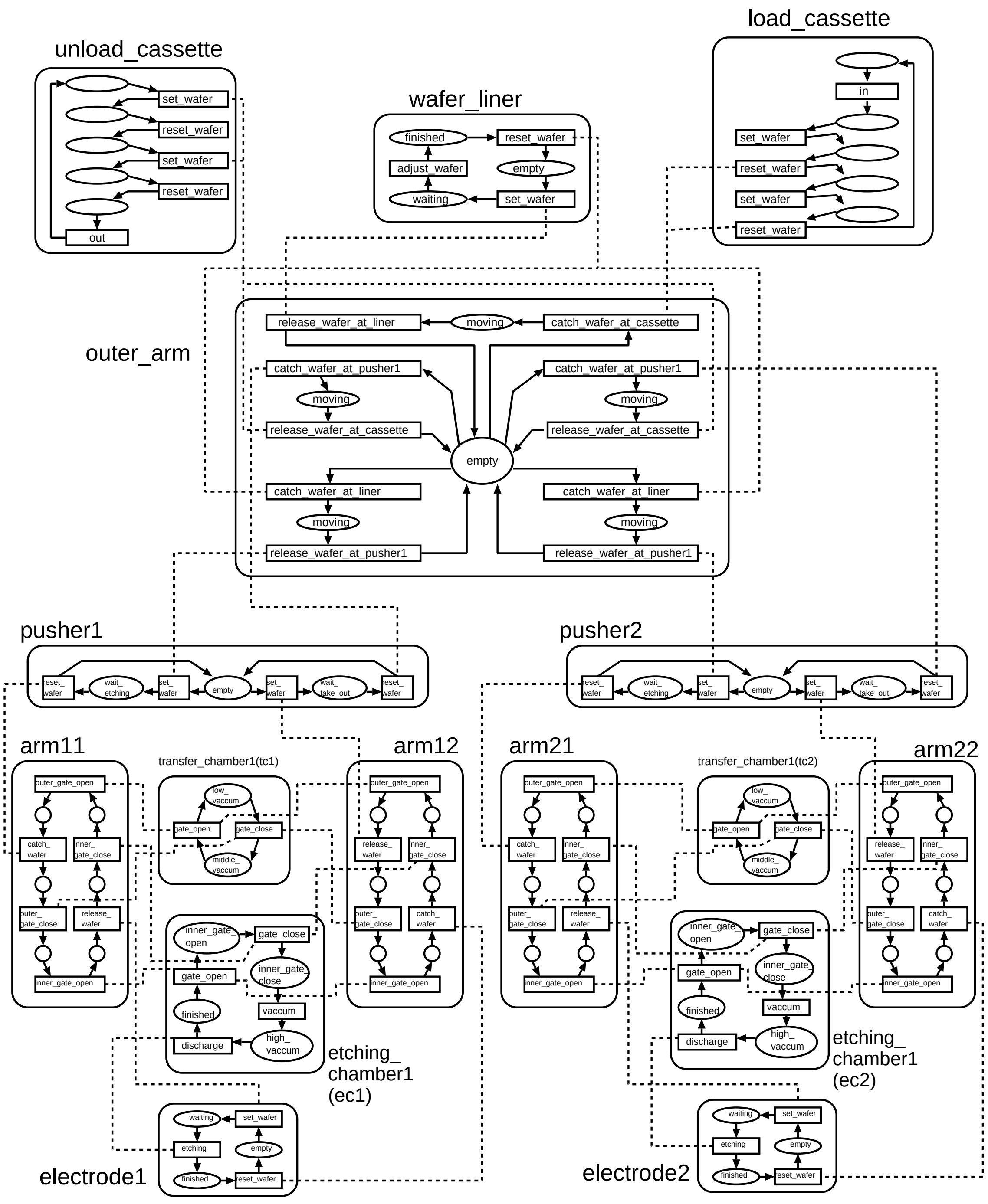


Figure 34. Machine Control Software (Transition Systems)

1. The complete deadlock never occurs.

$$(([],[]),\neg(epet\ internal\_deadlock))$$

2. Whenever an action *in* occurs, an action *out* eventually occurs. The action *in* means a material is put in a loading cassette (cassette1), and the action *out* means a material is got out of an unloading cassette (cassette2).

$$(([in,out],[]),apat([[in]]^-(apat(epet\ \ll out\gg^-))))$$

3. An inner gate and an outer never become open at the same time to keep vacuum condition better in an etching chamber.

$$(([],[tc1(outer\_gate\_open),ec1(inner\_gate\_open)]),apat\neg(tc1(outer\_gate\_open)\wedge ec1(inner\_gate\_open)))$$

$$(([],[tc2(outer\_gate\_open),ec2(inner\_gate\_open)]),apat\neg(tc2(outer\_gate\_open)\wedge ec2(inner\_gate\_open)))$$

Figure 9 shows the effectiveness of reduction in the first verification query ("the complete deadlock never occurs").

Table 9. Machine Control Software (Effect of Reduction)

Transition System	*Final Size*	*Maximum Temporary Size*
machine (original)	16741	16741
$machine_{red}$(reduction function (1))	4	1276
$machine_{red}$ (reduction function (2))	1425	4218

In this example, only synchronization parts of the system are modeled by transition systems. Actually, they have a lot of non-synchronization (functional) parts which are unrelated to verification queries. Therefore, the reduction will be more effective.

5 Toward Practical Verification

5.1 Translation Target Program to Transition Systems

How does the designer model the target systems with transition systems in the practical software development? Since a transition system is a very simple model, it is impractical to describe the systems using naive transition systems. In fast most reactive and concurrent systems can be modeled as extended finite state transition systems, such as *Petri net*, *Statechart*, *HMS machine* [Gabrielian 91], and *SFC* (Sequential Function Chart) [IEC 1131-3]. Generally speaking, it is possible to translate them to naive transition systems with some approximation.

This section presents an example of applying our compositional verification method to actual chemical plant control systems, in which control programs written by SFC are automatically translated into transition systems. SFC is a kind of safe Petri net which has shared memories. In the latter chapter, we will also provide high-level Petri nets (MENDEL nets) to describe the target reactive and concurrent systems, from which transition systems can be retracted as skeletons of them.

5.2 Chemical Plant Control Software

This section shows how our verification method can be applied to practical chemical plant control systems. The point of this application is how to translate a practical plant control system to a set of transition systems. We briefly illustrate the translation and validation process using an example of a simple chemical plant shown in Fig. 35.

A requirement for a control software of this plant is described as follows. Two raw materials are poured into a reactor via pipe1 and pipe2. When the level sensor of the reactor indicates high, valve1 and valve2 are closed. Then, valve4 opens and hot steam is poured into the reactor via pipe4 until

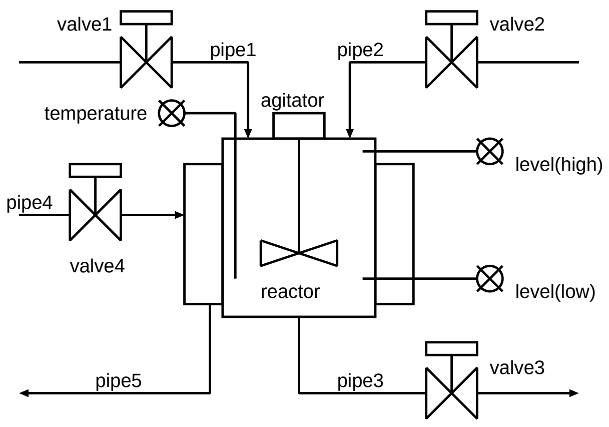


Figure 35. Example of Chemical Plant

the temperature of the reactor becomes high. At high temperature, chemical reaction is caused by an agitator. Finally, produced materials are extracted from the reactor via pipe3 by valve3.

In this example, the control software consists of two processes; the main control process ($task_1$) and the temperature control process ($task_2$) [10] . These tasks are described by *SFC* which is a popular graphical programming language for sequence control systems (Fig. 36 and Fig. 37).

When applying the compositional verification method to the plant control software, the target system should be translated to a set of transition systems. The target system consists of the control software and controlled objects (plant equipment), both of which should be translated to transition systems because the control software can not function without connecting it with controlled objects. We briefly show procedures of the translation.

- **SFC Tasks → Transition Systems**

  In translation from a SFC task to transition system, two issues have to be considered; concurrency within a task and access to shared memories.

  - Concurrency within a task

    Each SFC task is translated to a transition system. Since SFC is based on Petri nets, SFC may have concurrency in a task. For example, `v1_open` and `v2_open` are concurrently processed in Fig. 36. If a SFC task has no concurrency, then a generated transition system almost corresponds to a SFC task by one to one. If a SFC task has concurrency, then a global transition system is produced which is equivalent to the original SFC by interleaving concurrent actions. Note that this interleaving within a task has possibility to cause state explosion, but it usually does not because most state explosions are caused by interleaving among concurrent tasks and not by interleaving within a task.

  - Access to shared memories

    In plant control systems, communication among tasks and communication between tasks and plant are done by way of shared memories. It means *asynchronous* communication. As a transition system supposes only *synchronous* communication, access to shared memories have to be translated into *synchronous* communication. In our method, possible access patterns of a shared memory are modeled as actions, and each access is interpreted as synchronization of the same actions. For example, figure 39 illustrates how to translate accesses to shared memories for communication among task1 and valve1.

  Figure 38 shows a generated transition system of `task1`.

- **Plant Modeling with Transition Systems**

[10] Here, "task" and "process" are identical. Since "task" is conventionally used in plant control systems, we use "task" in the following explanation.

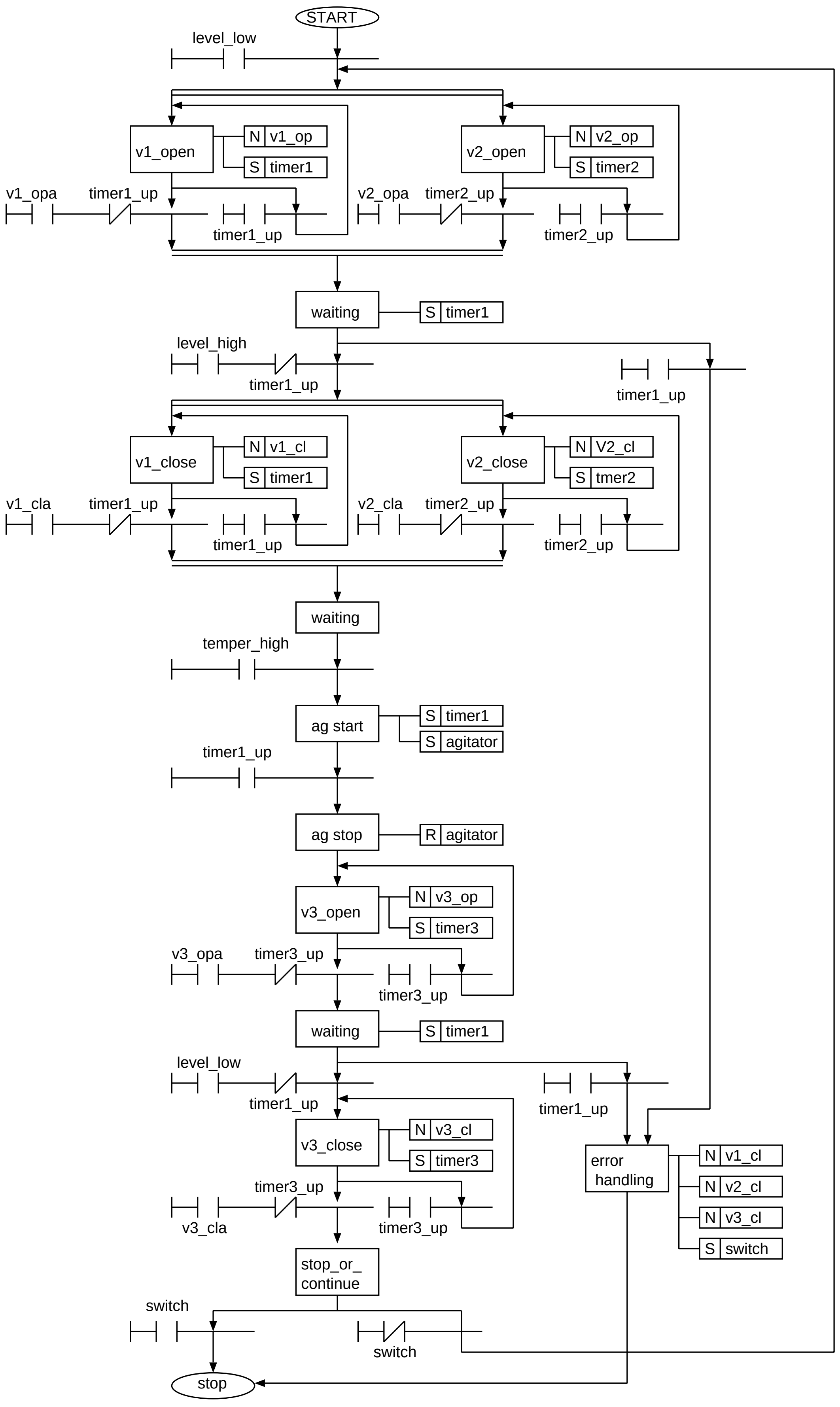


Figure 36. Plant Control Software (task1)

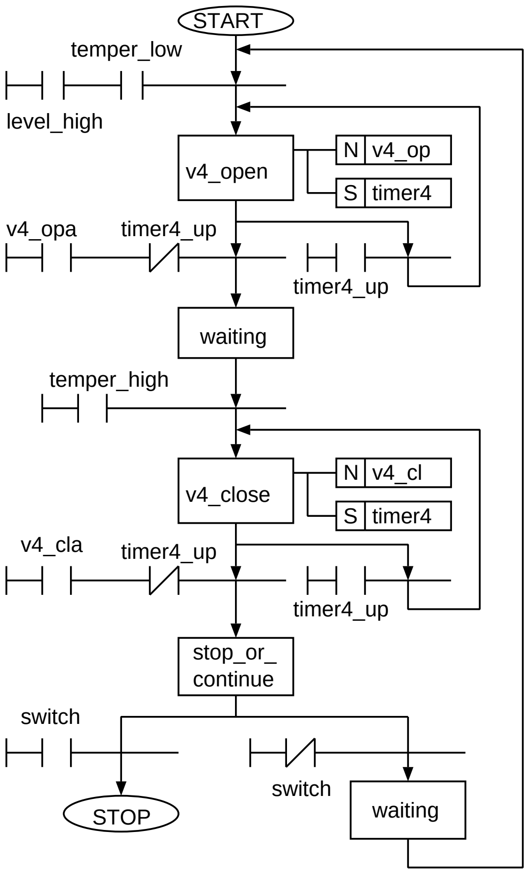


Figure 37. Plant Control Software (task2)

Controlled objects should be directly modeled by the designer with transition systems. This may be a tedious job. Focusing on chemical plants, we have proposed a method to generate automatically transition systems (plant simulator) by reusing and connecting model fragments corresponding to plant equipment from plant configuration data (e.g., process flow diagram) [Kawata 95, ?].

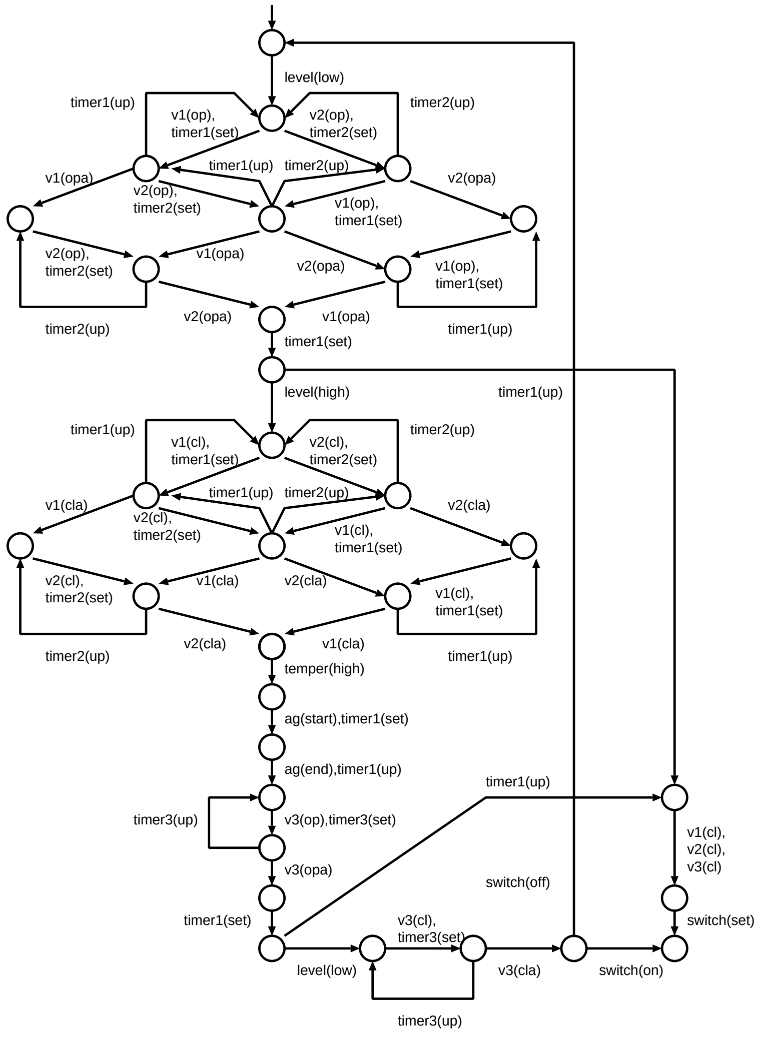


Figure 38. Transition System (task1)

Figure 40 shows a final structure of the plant control system components of which are modeled by transition systems.

Verification queries for this plant control software are listed as follows.

- There exists no complete deadlock except normal termination $(task1(stop) \wedge task2(stop2))$.
  $(([], [task1(stop), task2(stop2)]), apat(internal\_deadlock \supset (task1(stop) \wedge task2(stop2))))$
- The normal termination state is reachable from every state which is reachable from an initial state.
  $(([], [task1(stop), task2(stop2)]), apat(epet\ (task1(stop) \wedge task2(stop2))))$
- There are no abnormal states such that material pouring and extracting are simultaneously carried out in the reactor.
  $(([], [valve1(open), valve2(open), valve3(open)]), apat\ \neg((valve1(open) \vee valve2(open)) \wedge valve3(open)))$
- There are no dangerous states such that a reactor is heated up although material is not filled up enough.
  $(([], [valve4(open), level(high)]), apat\ valve4(open) \supset level(high))$

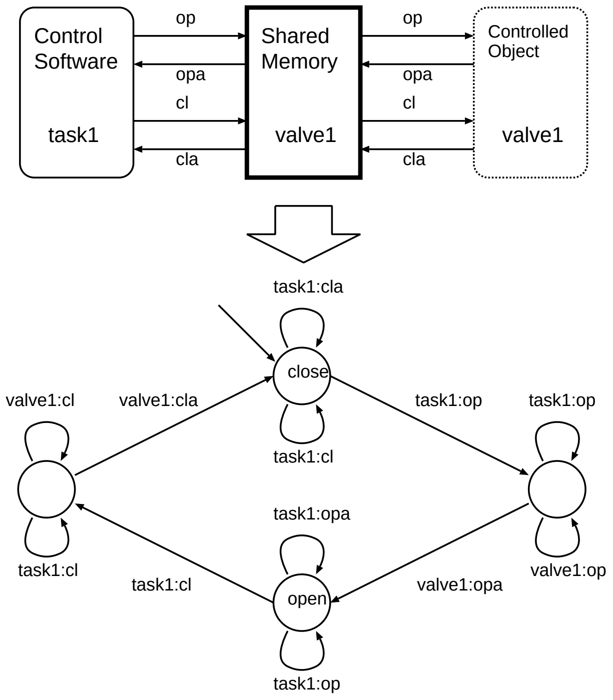


Figure 39. Modeling of Shared Memory

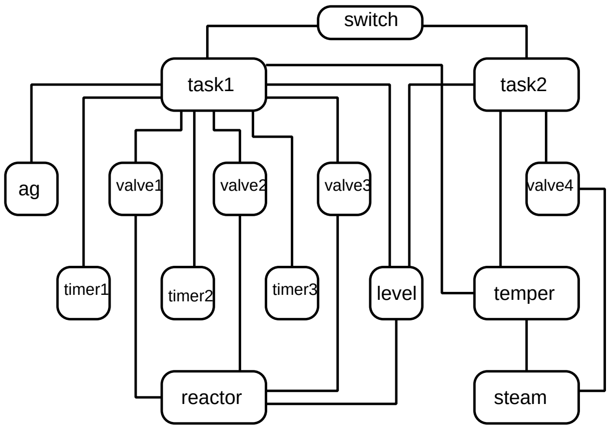


Figure 40. Structure of the Plant Control System

In fact, This SFC program has a lot of bugs, most of which can be detected as complete deadlock except normal termination. To put it concretely, there exist the following bugs.

- When a termination switch is turned on in certain timing, it is possible that one of tasks is terminated and the other is hung up.
- Since task2 can not completely cope with error handling of task1, task2 is possibly hung up.
- When it takes long time to close valve1 and valve2, heating process of task2 may be finished. On that occasion, task1 may be hung up by waiting forever the temperature becomes high.

The effectiveness of computing cost in compositional verification in the case of deadlock detection is shown in Table 10.

Table 10. State Space Generation (Chemical Plant Control System)

Method	*States*	*Transitions*
Compositional Method	528	4064
Naive State Space Generation	283439	2174168

5.3 Verification Tool

We initially implemented primitive operations used in the compositional verification including composition, relabeling, reduction, and PQL model checker. However, when the verification method is actually applied, the following problems still remain.

- It is not necessarily easy for ordinary designers to describe in PQL queries.
- The exact location of bugs cannot be spotted even if the existence of bugs can be detected.
- Process composition and relabeling operations are laborious.

Therefore, we have developed a took kit named "*VERASQUES*" (VERificAtion Systems for temporal logic QUErieS) which is of general purpose and includes the following facilities to solve the above problems.

- Primitive operation commands
- PQL query generation interface
- Debugger which supports to identify the location of bugs
- PCL (Process Composition Language) which is a command language to describe operation procedures.

When focusing on certain domains such as chemical plants, the more specific verification tool can be available. *SAVE/SFC* [Uchihira 93a, Kawata 96] is a Simulation And Verification Environment for SFC programs which we have been now developing (Fig. 41). SAVE/SFC has a simulation facility in addition to the verification facility. Moreover, verification is tightly tuned up by merging compositional method and partial order method [Uchihira 95b]and introducing a lot of heuristics.

We have applied SAVE/SFC to a real chemical plant control software, where the plant consists of 28 valves, 4 pumps, 5 measurements, and 1 reactor, and the program consists of 5 tasks. Many trivial bugs can be found by typical and varied simulation using the heuristic selection. Some complicated bugs that are difficult to find by typical simulation have been detected by exhaustive simulation using the logical selection. The verification generated the reduced state space which has 2800 states, and takes about 3 hours.

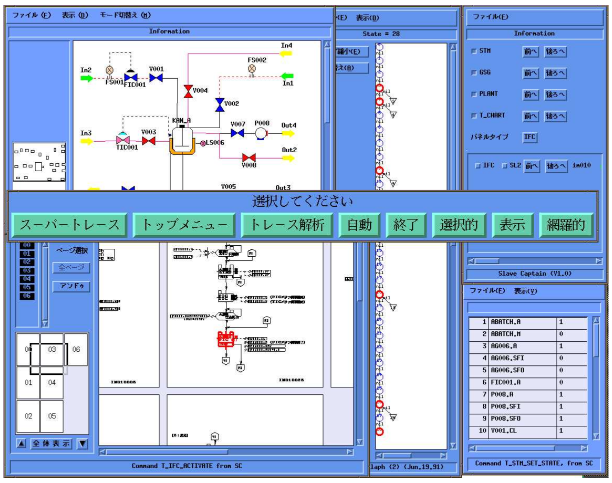


Figure 41. SAVE/SFC

6 Related Works

6.1 Compositional Verification Methods

Many *compositional verification* methods have been proposed for a system which has compositional structure like CCS, CSP, and Modular Petri nets. Generally speaking, compositional verification can be defined as

- verifying properties of the components of a system, and then
- deducing global properties from these local properties.

This compositional approach can reduce verification cost drastically. Compositional verification should be compositional in the structure of processes and work purely on the syntactical level without inspecting components. An *ideal* method of compositional verification can be formalized as follows.

$$(op(T_1, T_2, ..., T_i, ..., T_n) \models f) \iff$$
$$op_f((T_1 \models op_1^{-1}(f)), (T_2 \models op_2^{-1}(f)), ..., (T_i \models op_i^{-1}(f)), ..., (T_n \models op_n^{-1}(f)))$$

where $T_1, T_2, ..., T_i, ..., T_n$ are component processes of the system and $op(T_1, T_2, ..., T_i, ..., T_n)$ is a composite process using an operator op over the components, f is a formula representing verification queries, $T \models f$ means a process T satisfies f, $op_i^{-1}(f)$ is a formula which is derived from f and op for i-th argument of op, op_f is a logical operator corresponding to op. Here, $op_i^{-1}(f)$ is a projection of f onto i-th argument of op. Note that $op_i^{-1}(f)$ is derived without inspecting components $T_1, T_2, ..., T_n$. This function op_i^{-1} plays an important role in the compositional verification, that is, if $T_i \models op_i^{-1}(f)$ can be locally verified for every component T_i, then $op_f((T_1 \models op_1^{-1}(f)), (T_2 \models op_2^{-1}(f)), ..., (T_n \models op_n^{-1}(f)))$ can be computed instead of globally verifying $op(T_1, T_2, ..., T_i, ..., T_n) \models f$.

For example, a summation operator "+" of CCS is good-natured for the compositional verification. The compositional verification with regard to "+" can be formalized as follows.

$$(T_1 + T_2) \models \langle a \rangle f \iff (T_1 \models \langle a \rangle f) \vee (T_2 \models \langle a \rangle f),$$

where $op = +$, $op_i^{-1}(\langle a \rangle f) = \langle a \rangle f$ and $op_{\langle a \rangle f} = \vee$. In this case, it is sufficient to prove local properties $(T_1 \models \langle a \rangle f)$ and $(T_2 \models \langle a \rangle f)$ instead of proving a global property $(T_1 + T_2) \models \langle a \rangle f$.

Unfortunately, it is proved that it is impossible to find $op_i^{-1}(f)$ for every operator op of CCS [Fantechi 91]. In particular, a composition operator is inadequate for the ideal compositional verification. Therefore, several *non-ideal* compositional verification methods have been proposed with avoiding this essential limitation.

1. Restriction of verification queries
   For example, Winskel [Winskel 90] showed a compositional verification for restricted assertions, in which an assertion over process composition (product) $T_1 \times T_2$ should be described as product of assertions $A_1 \times A_2$.

2. Introduction of component's information to $op_i^{-1}(f)$
   If we are allowed to inspect components $T_1, ..., T_n$ and use an extended projection $op_i^{-1}(f, T_1, ..., T_n)$ instead of $op_i^{-1}(f)$, pseudo-compositional verification is available. Since the projection op_i^{-1} does not require entire information of $T_1, ..., T_n$, it is possible to derive a relatively simple and equivalent formula $red(op_i^{-1}(f, T_1, ..., T_n))$ by apply reduction rules to the original $op_i^{-1}(f, T_1, ..., T_n)$. Andersen and Winskel [Andersen 92] propose a compositional verification method based on this framework.

The latter approach is essentially equivalent to our and Clarke's compositional verification methods. The difference between them is concerned with the place on which information derived by inspecting components is reflected. In our and Clarke's methods, $T_i^{+\alpha} \models op_i^{-1}(f)$ is used instead of $T_i \models op_i^{-1}(f)^{+\alpha}$ in verifying local properties, where

$$T_i^{+\alpha} = op(T_1, ..., T_i, ..., T_n)$$

$$op_i^{-1}(f)^{+\alpha} = op_i^{-1}(f, T_1, ..., T_n).$$

In other word, the information $(+\alpha)$ is reflected on a transition system T_i in our and Clarke's methods, while it is reflected on a formula $op_i^{-1}(f)$ in Andersen and Winskel's method. Note that both methods make good use of reduction techniques. More specifically, we can show the following correspondence for the formulation of our compositional verification method described in Section 4.2.

$$\begin{aligned}(T_1 \mid T_2 \models f_1 \wedge f_2) &\iff \\ &((T_{f_1} \models f_1) \wedge (T_{f_2} \models f_2))\end{aligned}$$

where $T_{f_1} = T_1^{+\alpha} = red((red(T_1[l_{11}]) \mid red(T_2[l_{12}]))[l_{VS_1}])$ and $T_{f_2} = T_2^{+\alpha} = red((red(T_1[l_{21}]) \mid red(T_2[l_{22}]))[l_{VS_2}])$

Although Andersen and Winskel's approach is essentially equivalent to our and Clarke's methods, our approach is more appropriate for transition systems which can utilize (extended) bisimulation for process reduction.

6.2 Comparison with Partial Order Approach

Another approach to avoid the *state explosion* is *partial order approach* [Valmari 90, Godefroid 91a, Godefroid 91b, Wolper 93, Godefroid 96]. When there are many redundant interleaving simulation paths over the state space, we can leave one representative path and delete the others by the *partial order method*. Figure 42 illustrates an essential idea of the partial order method. A naive state space of $P_1 \mid P_2$ has 18 states and 20 paths by transition interleaving (i.e., total ordering). Here, P_1 and P_2 synchronize only about actions a and b. However, if P_1 and P_2 are independent regarding actions $t_{11}, ..., t_{23}$, this naive state space is redundant in order to detect deadlock. It is sufficient to analyze only one representative path $a \rightarrow t_{11} \rightarrow t_{12} \rightarrow t_{13} \rightarrow t_{21} \rightarrow t_{22} \rightarrow t_{23} \rightarrow b$ which consists of 9 states.

To compare the partial order method with the compositional method, we show Fig. 43 where $t_{11}, t_{13}, t_{21}, t_{23}$ are internal (τ) actions and processes P_1 and P_2 are first reduced and then composed. This method can reduce 18 states of a naive state space to 7 states.

These two approaches can be characterized as follows.

- **Partial Order Method**
  - Compatibility with conventional testing and simulation
    Partial order method is well-suited to actual execution of target programs because a selection mechanism of a representative path can be implemented as a smart scheduler. Verification is modeled as a smart exhaustive simulation and harmonized with conventional testing and simulation. In other words, conventional testing and simulation are interpreted as a special case of a smart exhaustive simulation based on the partial order method.

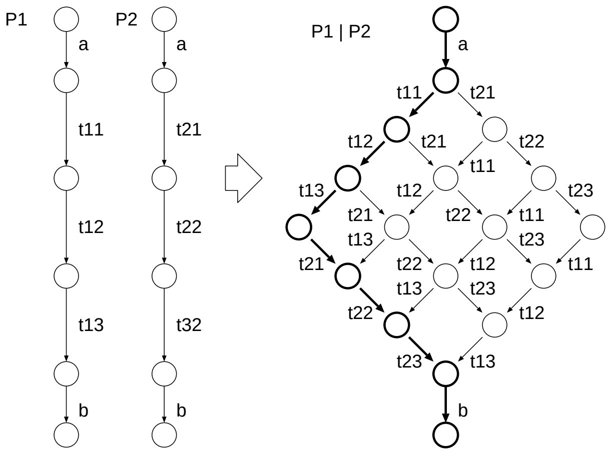


Figure 42. Partial Order Method

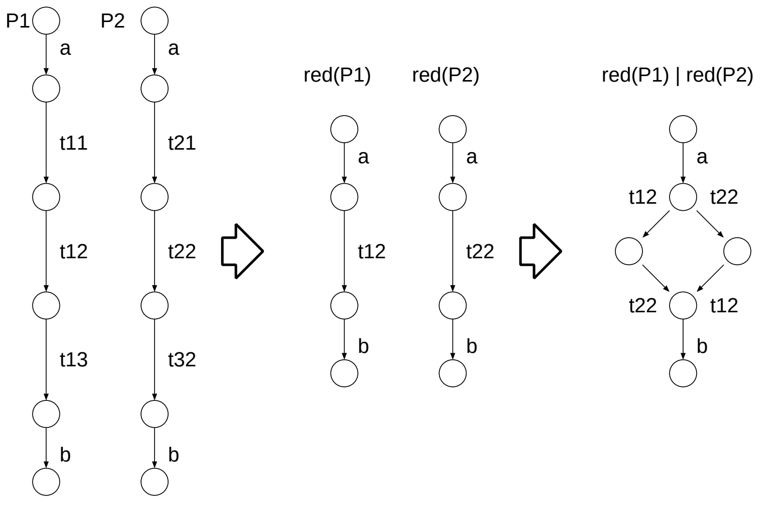


Figure 43. Compositional Method

– Applicability to ill-structured systems

  It is not so difficult to implement a smart scheduler which requires only information about data and control dependency between processes. Therefore, it is applicable to ill-structured systems.

– Flat and global analysis

  Since the partial order method is a smart version of global state apace analysis, it can not make good use of hierarchical and compositional structures of the target systems.

- **Compositional Method**

  – Effectiveness for well-structured systems

    The compositional method is effective for well-structured systems which consist of many but uniform and mostly independent subprocesses because process reduction works well for them.

  – Hierarchical and compositional analysis

    The compositional method is effective for large but hierarchical systems because verification is done compositionally where subprocesses are abstracted by process reduction.

Since compositional methods and partial order methods are orthogonal, it is practical solution to utilize both methods complementarily (hybrid method). For example, we illustrate a typical hybrid method for the target system which consists of two large processes P_1 and P_2 (Fig. 44). First P_1 and P_2 is reduced to simple interface processes which produces only behaviors related to given queries and synchronization by the compositional method, then a smart state space generation is done by the partial order method. Finally, the state space is analyzed by PQL model checker. In fact we adopted this hybrid approach in SAVE/SFC [Uchihira 93a, Uchihira 95b] which is mentioned in the previous section.

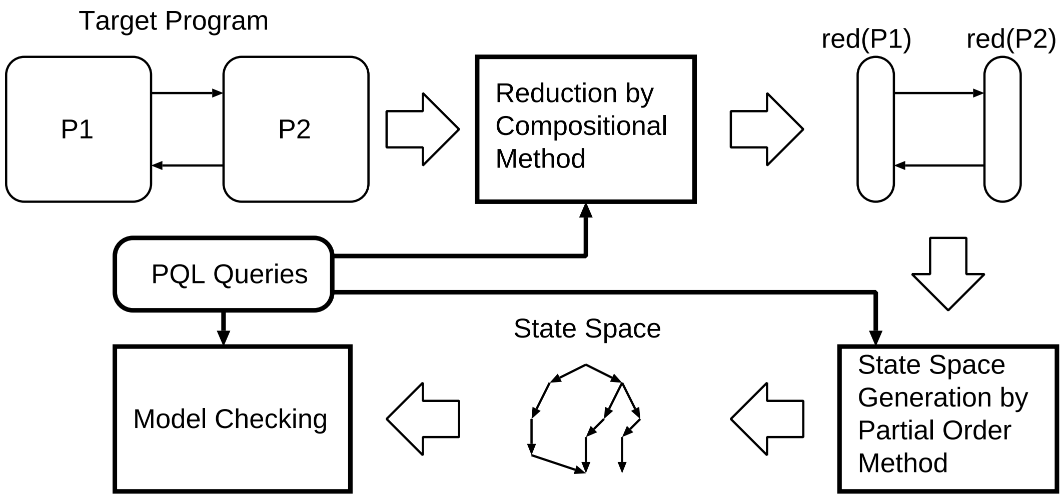

Target Program
P1
P2
Reduction by Compositional Method
red(P1)
red(P2)
PQL Queries
State Space
Model Checking
State Space Generation by Partial Order Method

Figure 44. Hybrid Approach of Compositional and Partial Order Methods

7 Summary

Process Query Language of concurrent program verification based on state logic and the compositional verification method were proposed and its effectiveness confirmed by using the examples.

This chapter focuses on the compositional verification for transition systems, which have only finite states, in place of Petri nets, which may have infinite states. A model-checking method for Petri nets was proposed by [Bradfield 92], which is very powerful but not compositional (he showed some remarks on compositionality at page 82 of [Bradfield 92]). In general the compositional verification for Petri nets is difficult and impractical. Therefore, developing a compositional verification method for Petri nets, which may be restricted to some degree but practical enough, is one of the further works.

Appendix I : Proof of Theorem 10

To proof theorem 10, several definitions and lemmas are introduced.

Definition 29 ($\approx^k_{\pi\tau\omega}$)
In (S, Act, δ, P, π), *for* $\forall s, t \in S, \forall k \geq 0$,
$s \approx^0_{\pi\tau\omega} t \overset{def}{=}$

- $\pi(s) = \pi(t)$
- $s \uparrow$ *iff* $t \uparrow$

$s \approx^{k+1}_{\pi\tau\omega} t \overset{def}{=}$

- $\pi(s) = \pi(t)$
- $\forall a.\forall s'.(\mathit{if}\ s \xrightarrow{a} s'\ \mathit{then}\ \exists t'.t \overset{\hat{a}}{\Rightarrow} t' \wedge s' \approx^k_{\pi\tau\omega} t')$
- $\forall a.\forall t'.(\mathit{if}\ t \xrightarrow{a} t'\ \mathit{then}\ \exists s'.s \overset{\hat{a}}{\Rightarrow} s' \wedge s' \approx^k_{\pi\tau\omega} t')$
- $s \uparrow$ *iff* $t \uparrow$

Lemma 7 (Relation of $\approx_{\pi\tau\omega}$ and $\approx^k_{\pi\tau\omega}$)

(1)
$$s \approx_{\pi\tau\omega} t \Leftrightarrow \bigwedge_{k=1}^{\infty} (s \approx^k_{\pi\tau\omega} t)$$

(2)
$$s \not\approx_{\pi\tau\omega} t \Leftrightarrow \bigvee_{k=1}^{\infty} (s \not\approx^k_{\pi\tau\omega} t) \Leftrightarrow \exists k.(s \not\approx^k_{\pi\tau\omega} t)$$

Proof. It is obvious from the definition.□

Lemma 8 (Semantics of $\ll\ \gg^- f, \ll\ \gg^+ f, \ll a \gg^- f, \ll a \gg^+ f$)

(1)
$$s \models \ll\ \gg^- f \iff \exists t = \tau^*.\exists s'.(s \xrightarrow{t} s' \wedge \underline{s'} \models f)$$

(2)
$$s \models \ll\ \gg^+ f \iff \exists t = \tau^*.\exists s'.(s \xrightarrow{t} s' \wedge \underline{s'} \models f) \vee \exists t = \tau^\omega.(s \xrightarrow{t}\ )$$

(3)
$$s \models \ll a \gg^- f \iff \exists t = \tau^* a \tau^*.\exists s'.(s \xrightarrow{t} s' \wedge \underline{s'} \models f)$$

(4)
$$s \models \ll a \gg^+ f \iff \exists t = \tau^* a \tau^*.\exists s'.(s \xrightarrow{t} s' \wedge \underline{s'} \models f) \vee \exists t = \tau^* a \tau^\omega.(s \xrightarrow{t}\ ) \vee \exists t = \tau^\omega.(s \xrightarrow{t}\ )$$

Here $\exists t = \tau^* a \tau^*$ *means* $\exists t \in \{\tau^i a \tau^j \mid i, j \geq 0\}$.

Proof.

(1) From lemma 6, $s \in V[\![\ll\ \gg^- f]\!] = [\![\mu Z.(f \vee \exists TZ)]\!] \iff \exists k.s \in V[\![(\lambda Z.(f \vee \exists TZ))^k false]\!] = V[\![(\exists T)^{k-1} f]\!] \iff \exists t = \tau^*.(s \xrightarrow{t} s' \wedge s' \in V[\![f]\!])$.

(2) From lemma 6, $V[\![\ll\ \gg^+ f]\!] = V[\![\nu Z.(f \vee \underline{\exists} TZ)]\!] = \lim_{k\to\infty} V[\![\lambda Z.(f \vee \exists TZ)^k true]\!] = \lim_{k\to\infty} \bigcup_{i=0}^{k-1} V[\![(\exists T)^i f]\!] \cup V[\![(\exists T)^k true]\!]$. Therefore, $s \in V[\![\ll\ \gg^+ f]\!] \iff \exists i.(s \in V[\![(\exists T)^i f]\!]) \vee s \in V[\![(\exists T)^\omega true]\!] \iff \exists t = \tau^*.(s \xrightarrow{t} s' \wedge s' \in V[\![f]\!]) \vee t = \tau^\omega.(s \xrightarrow{t}\ )$.

(3) In the same way as (1), $s \in V[\![\ll a \gg^- f]\!] = V[\![\mu Z_1.(\exists(a \wedge X(\mu Z_2.(f \vee \exists TZ_2))) \vee TZ_1)]\!] \iff \exists i,j.s \in V[\![(\exists T)^i \exists(a \wedge X(\exists T)^j f)]\!]$. Therefore, $s \in V[\![\ll a \gg^- f]\!] \iff \exists t = \tau^* a \tau^*.(s \xrightarrow{t} s' \wedge s' \in V[\![f]\!])$.

(4) In the same way as (2), $V[\![\nu Z.(f \vee \exists TZ)]\!] = \lim_{k\to\infty}(\bigcup_{i=0}^{k-1} V[\![(\exists T)^i f]\!] \cup V[\![(\exists T)^k true]\!])$. Therefore, $s \in V[\![\ll a \gg^+ f]\!] = V[\![\nu Z_1.(\exists(a \wedge X(\nu Z_2.(f \vee \exists TZ_2))) \vee TZ_1)]\!]$
$\iff \exists i,j.(s \in V[\![(\exists T)^i \exists(a \wedge X(\exists T)^j f)]\!]) \vee \exists i.(s \in V[\![(\exists T)^i \exists(a \wedge X(\exists T)^\omega true)]\!]) \vee s \in V[\![(\exists T)^\omega true]\!]$.
Then, $s \in V[\![\ll a \gg^+ f]\!] \iff \exists t = \tau^i a \tau^j.(s \xrightarrow{t} s' \wedge \underline{s'} \in V[\![f]\!]) \vee \exists t = \tau^i a \tau^\omega.(s \xrightarrow{t}\ ) \vee \exists t = \tau^\omega.(s \xrightarrow{t}\ )$.
□

Proof of Theorem 10

[Proof of $T_1 \approx_{\pi\tau\omega} T_2 \ \Leftarrow\ T_1 \approx_{PQL} T_2$**]**
If $T_1 \not\approx_{\pi\tau\omega} T_2$, it is sufficient to prove $\exists f.(T_1 \models f \wedge T_2 \not\models f)$. As $\exists k.(s_{01} \not\approx^k_{\pi\tau\omega} s_{02})$ from lemma 7, the induction about k can be applied as follows. In the case of $s \not\approx^k_{\pi\tau\omega} t$, f such that $s \models f \wedge t \not\models f$ can be constructed for the following 4 cases.

Case 1 $\pi(s) \neq \pi(t) \ \Rightarrow\ f = p\ s.t.\ p \in \pi(s), p \notin \pi(t)$.

Case 2 $s\uparrow \wedge \neg(t\uparrow) \ \Rightarrow\ f = \ll\ \gg^+ false$. (from Lemma 8)

Case 3 $s \xrightarrow{a} s' \wedge \neg\exists t'.(t \overset{\hat{a}}{\Rightarrow} t') \ \Rightarrow\ f = \ll a \gg^- true$. (from definitions)

Case 4 $s \xrightarrow{a} s' \wedge \forall t'_i.(t \overset{\hat{a}}{\Rightarrow} t' \Rightarrow s' \not\approx^{k-1}_{\pi\tau\omega} t'_i) \ \Rightarrow$
By the induction,
$\forall t'_i.\exists f_i.(s' \models f_i \wedge t'_i \not\models f_i)$. Then, $f = \ll a \gg^- \bigwedge_i f_i$.

[Proof of $T_1 \approx_{\pi\tau\omega} T_2 \ \Rightarrow\ T_1 \approx_{PQL} T_2$**]**
$\forall f.(T_1 \approx_{\pi\tau\omega} T_2 \wedge T_1 \models f \ \Rightarrow\ T2 \models f)$ is shown here. From Lemma 6(3), only finite length PQL formulas with no μ operators should be considered. When PQL formulas which do not contain μ operators, it can be proved by the structural induction of PQL formulas as follows. Here the cases of $f = \ll a \gg^+ f', f = \ll a \gg^- f'$ are proved. As for other formulas, the proof can be provided in the same manner.

- $f = \ll a \gg^- f'$:
  When $s_1 \approx_{\pi\tau\omega} s_2$ and $s_1 \models \ll a \gg^- f'$, $\exists t_1 = \tau^* a \tau^*.(s_1 \xrightarrow{t_1} s'_1 \wedge s'_1 \models f)$ from Lemma 8, also $\exists t_2 = \tau^* a \tau^*.(s_2 \xrightarrow{t_2} s'_2 \wedge s'_1 \approx_{\pi\tau\omega} s'_2)$ from $s_1 \approx_{\pi\tau\omega} s_2$. By the structural induction, $s'_2 \models f'$, then $s_2 \models \ll a \gg^- f'$.

- $f = \ll a \gg^+ f'$:
  When $s_1 \approx_{\pi\tau\omega} s_2$ and $s_1 \models \ll a \gg^+ f'$, from Lemma 8,

  (1) $\exists t = \tau^* a \tau^*.(s_1 \xrightarrow{t} s'_1 \wedge s'_1 \models f)$, or

  (2) $\exists t = \tau^\omega.(s \xrightarrow{t}\ )$, or

  (3) $\exists t = \tau^* a \tau^\omega.(s \xrightarrow{t}\ )$.

  In the case of (1), it can be proved in the same manner as $f = \ll a \gg^- f'$. In the case of (2), since $s_1\uparrow$ and $s_1 \approx_{\pi\tau\omega} s_2$, $s_2\uparrow$. Therefore $s_2 \models \ll a \gg^+ f'$. In the case of (3), since $\exists t_1 = \tau^* a \tau^*.(s_1 \xrightarrow{t_1} s'_1 \wedge s'_1\uparrow)$ and $s_1 \approx_{\pi\tau\omega} s_2$, $\exists t_2 = \tau^* a \tau^*.(s_2 \xrightarrow{t_2} s'_2 \wedge s'_2\uparrow)$. Therefore, $s_2 \models \ll a \gg^+ f'$ □

Appendix II : Well-known Equivalence Relations and Divergence

This section shows detail definitions of well-known equivalence relations (trace equivalence, failure equivalence, partial bisimulation equivalence) referred in Fig. 28. Their relations are summarized from the viewpoint of "divergence".

Definition 30 (trace equivalence)
Let S be a set of states, A be a set of actions ($Act = A \cup \{\tau\}$), and $\delta : S \times Act \to 2^S$ be a nondeterministic transition function. For (S, Act, δ) and $s, t \in S$, s and t are trace equivalent, written $s \approx_1 t$, if $\forall \theta \in Act^*. s \overset{\hat{\theta}}{\Rightarrow}$ iff $t \overset{\hat{\theta}}{\Rightarrow}$

Definition 31 (failure)
For (S, Act, δ) and $s \in S$,

$$failures(s) \overset{def}{=} \{(\theta, L) \mid \theta \in Act^*, L \subset A \text{ such that } \exists s' \in S.(s \overset{\hat{\theta}}{\Rightarrow} s' \text{ and } s' \overset{\tau}{\not\rightarrow} \text{ and } \forall a \in L. s' \overset{a}{\not\rightarrow})\}$$

Definition 32 (failure equivalence)
For (S, Act, δ) and $s_1, s_2 \in S$, failure equivalence ($\approx_f$) is defined as follows.

$$s_1 \approx_f s_2 \iff failure(s_1) = failure(s_2)$$

Definition 33 (partial bisimulation preorder by global divergence)
For (S, Act, δ), a partial bisimulation preorder with global divergence $\sqsubseteq_g$ ($\subset S \times S$) is defined as the largest relation such that
if $\forall s, t \in S, s \sqsubseteq_g t$ implies

- $\forall a \in Act. \forall s' \in S.($*if* $s \overset{a}{\rightarrow} s'$ *then* $\exists t' \in S. t \overset{\hat{a}}{\Rightarrow} t' \wedge s' \sqsubseteq_g t')$
- *if* $\neg(s \uparrow)$ *then*
  - $\neg(t \uparrow)$
  - $\forall a \in Act. \forall t' \in S.($*if* $t \overset{a}{\rightarrow} t'$ *then* $\exists s' \in S. s \overset{\hat{a}}{\Rightarrow} s' \wedge s' \sqsubseteq_g t')$

Definition 34 (partial bisimulation equivalence by global divergence)
For (S, Act, δ) and $s, t \in S$,

$$s \approx_p^g t \iff s \sqsubseteq_g t \wedge t \sqsubseteq_g s.$$

$\approx_p^g$ is called partial bisimulation equivalence by global divergence.

Definition 35 (parameterized $\tau\omega$-divergence)
For (S, Act, δ) and $s \in S, a \in Act$,

$$s \uparrow a \overset{def}{=} s \uparrow \text{ or } \exists s'.(s \overset{\hat{a}}{\Rightarrow} s' \wedge s' \uparrow)$$

Definition 36 (partial bisimulation preorder by local divergence)
For (S, Act, δ), a partial bisimulation preorder by local divergence $\sqsubseteq$ ($\subset S \times S$) is defined as the largest relation such that
if $\forall s, t \in S, s \sqsubseteq t$ implies

- $\forall a \in Act. \forall s' \in S.($*if* $s \overset{a}{\rightarrow} s'$ *then* $\exists t' \in S. t \overset{\hat{a}}{\Rightarrow} t' \wedge s' \sqsubseteq t')$
- $\forall a \in Act.$ *if* $\neg(s \uparrow a)$ *then*
  - $\neg(t \uparrow a)$
  - $\forall t' \in S.($*if* $t \overset{a}{\rightarrow} t'$ *then* $\exists s' \in S. s \overset{\hat{a}}{\Rightarrow} t' \wedge s' \sqsubseteq t')$

Definition 37 (partial bisimulation equivalence by local divergence)
For (S, Act, δ) and $s, t \in S$,

$$s \approx_p t \iff s \sqsubseteq t \wedge t \sqsubseteq s.$$

$\approx_p$ is called partial bisimulation equivalence by local divergence.

Partial bisimulation equivalence by local divergence is sometimes simply called “partial bisimulation equivalence”. The discrimination ability of partial bisimulation equivalence by local divergence is the same as *Intuitionistic Hennessy-Milner Logic (IHML)* [Stirling 87].

Theorem 11 (Imvolvement Relation among Equivalence Relations)
The imvolvement relation among equivalence relations is shown as follows.

- $\approx_{\tau\omega} \subset \approx_p^g$
- $\approx_p^g \subset \approx_p$
- $\approx_p^g \subset \approx_f$
- $\approx_f \subset \approx_1$
- $\approx_p \subset \approx_1$

Here, “$R_1 \subset R_2$” means “$\forall s, t \in S$. if sR_1t then sR_2t”.

Proof. It is obvious from the definition.□

Figure 45 shows a counter example which shows there is no imvolvement relation bwtween $\approx_f$ and $\approx_p$. In this example, $T_1 \approx_p T_2$ but $T_1 \not\approx_f T_2$. On the other hand, $T_3 \not\approx_p T_4$ but $T_3 \approx_f T_4$.

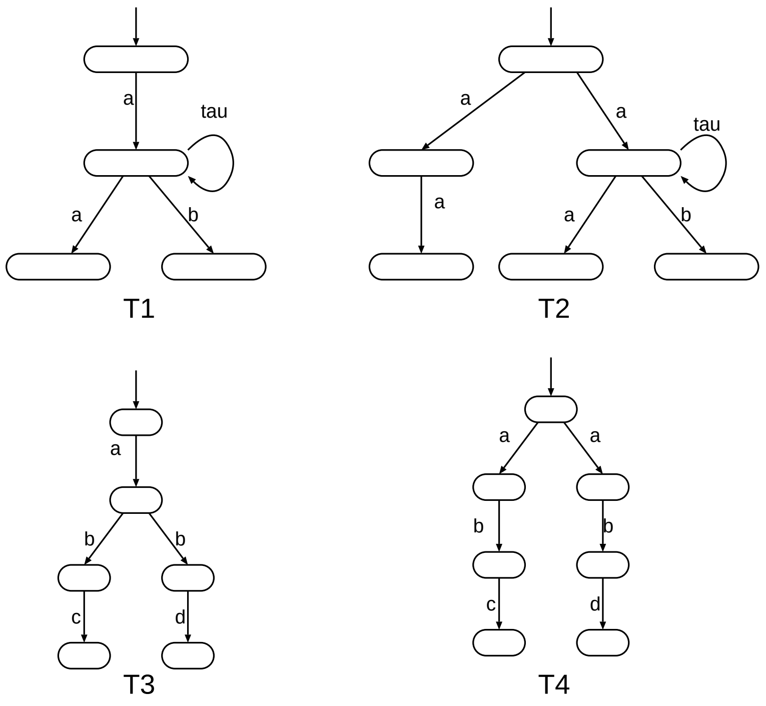


Figure 45. Example which shows there is no imvolvement relation bwtween $\approx_f$ and $\approx_p$

Chapter 6

Compositional Program Adjustment

In this chapter, we examine "*program adjustment*", a formal and practical approach to developing correct concurrent programs, by automatically adjusting an imperfect program to satisfy given constraints. A concurrent program is modeled by a finite state process, and program adjustment to satisfy temporal logic constraints is formalized as the synthesis of an arbiter process which partially serializes target (i.e. imperfect) processes to remove harmful nondeterministic behaviors. Compositional adjustment is also proposed for large-scale compound target processes, using process equivalence theory.

1 Motivation and Overview

1.1 Motivation

The difficulty of concurrent program debugging is mainly due to its nondeterministic behavior. We classify nondeterminism into the following 3 types.

- **Intended nondeterminism:** Nondeterministic behaviors which the programmer intends to implement.
- **Harmful nondeterminism:** Nondeterministic behaviors which the programmer does not intend to implement and does not expect.
- **Persistent nondeterminism:** Nondeterministic behaviors which have no effect on the results.

For example, Fig. 46 shows a simple Ada-like concurrent program "*Seat Booking*", where two processes read/write a shared memory "*seat*" to reserve one seat. This program has the 3 types of nondeterministic behaviors.

Intended nondeterminism The following nondeterministic behaviors θ_1 and θ_2 derive different results: P_1 can book the seat ($status_1 = OK$) in θ_1, but cannot ($status_1 = NG$) in θ_2. However both are correct (intended behaviors).

- $\theta_1 = l_1 \to l_2 \to l_3 \to l_4 \to l_5 \to m_1 \to m_2 \to m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG$.
- $\theta_2 = m_1 \to m_2 \to m_3 \to m_4 \to m_5 \to l_1 \to l_2 \to l_5$
  Result: $status_1 = NG, seat = OCCUPIED, status_2 = OK$.

Harmful nondeterminism The following nondeterministic behavior θ_3 derives an incorrect result (double booking). So, this program has harmful nondeterminism.
• $\theta_3 = l_1 \to m_1 \to l_2 \to m_2 \to l_3 \to m_3 \to l_4 \to m_4 \to l_5 \to m_5$
Result: $status_1 = OK, seat = OCCUPIED, status_2 = OK$.

Persistent nondeterminism The following two nondeterministic behaviors have the same result because l_1(write in $status_1$) and m_1(write in $status_2$) are independent actions of each other. We call such a situation *persistent.*

- $\theta_4 = \mathbf{l_1} \rightarrow \mathbf{m_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$
- $\theta_5 = \mathbf{m_1} \rightarrow \mathbf{l_1} \rightarrow l_2 \rightarrow l_3 \rightarrow l_4 \rightarrow l_5 \rightarrow m_2 \rightarrow m_5$
  Result: $status_1 = OK, seat = OCCUPIED, status_2 = NG.$

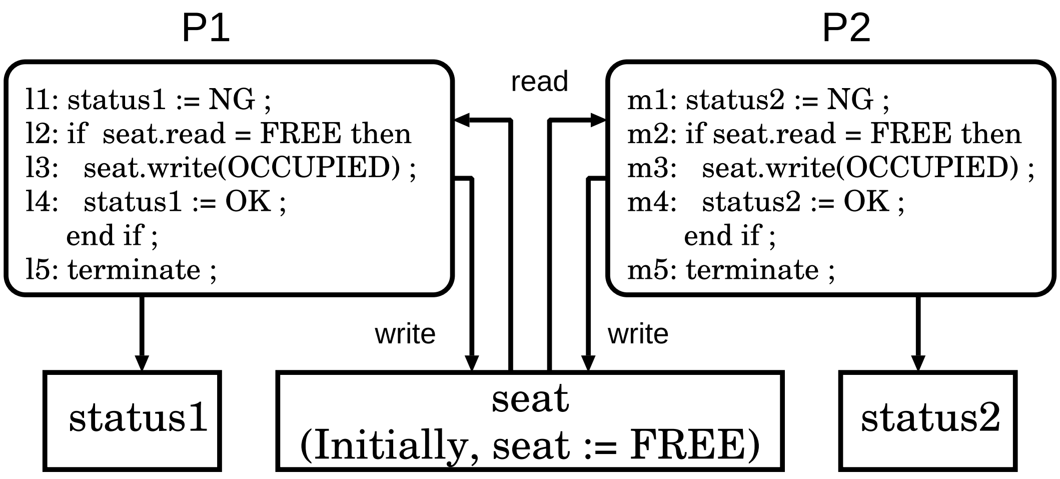


Figure 46. An example of a concurrent program

In our observation of concurrent program development, a programmer first tries to design and implement processes so as to maximize concurrency, which may include 3 types of nondeterminism. He then often finds harmful nondeterministic behaviors in testing and debugs them by partially serializing the critical sections which interfere each other using synchronization mechanisms (e.g. semaphores). Bugs due to harmful nondeterministic behaviors often account for a considerable part of all timing bugs.

We will show that the debugging processes for harmful nondeterministic behaviors can be mechanically supported using formal methods. It can be also regarded as a practical application of program synthesis techniques to program modification in debugging.

1.2 Overview of Main Results

We propose "program adjustment" which automatically adjusts (debugs) an imperfect program to satisfy given constraints. Here, we consider only timing constraints for concurrent programs that can be specified by temporal logic. In this context, "an imperfect program" is regarded as a program which is functionally correct but may be imperfect in its timing. We call such a program an *FCTI program* (Functionally-Correct Temporally-Imperfect program).

A concurrent program is modeled with the finite state process [Kanellakis 90] , which can specify the finite state transition system with liveness conditions. It can not only represent the transition systems in CCS [Milner 89], but also Büchi automata [Büchi 62]. A target FCTI program is compositionally constructed from several finite state processes with the composition operator "|"(ex. $P = (P_{11} \mid P_{12}) \mid (P_{21} \mid P_{22})$ in Fig.47(a)).

Basic Adjustment Program adjustment (*basic adjustment*) means to adjust an FCTI program to satisfy given constraints by adding an *arbiter* process which is synchronized with and restricts the behavior of the FCTI program (Fig.47(b)). The arbiter partially serializes the FCTI program to remove harmful nondeterministic alternatives which do not satisfy given constraints. We will show an algorithm to synthesize an arbiter process C_f automatically.

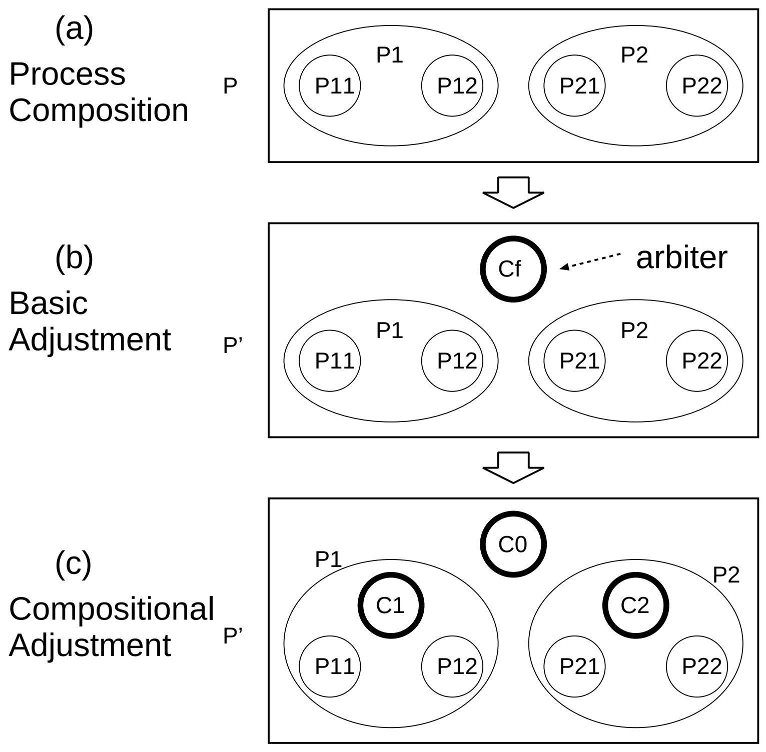


Figure 47. Process Composition (a), Basic (b), and Compositional (c) Adjustment

Input: An FTCI program P.

Input: Temporal logic constraints f.

Output: An arbiter process C_f such that $P \mid C_f$ satisfies f.

Compositional Adjustment When a target program becomes large, the arbiter synthesis may cause computing cost explosion. Therefore, we propose *compositional adjustment*, in which local arbiters are synthesized in each composition step. For example, an adjusted program with local arbiters C_0, C_1, and C_2 is shown as follows (Fig.47(c)).

$$P' = (P_{11} \mid P_{12} \mid C_1) \mid (P_{21} \mid P_{22} \mid C_2) \mid C_0$$

In each composition step, the reduction of the finite state process, based on process equivalence theory, can ease computing cost explosion. We introduce a new process equivalence relation ($\pi\tau\omega$-bisimulation) to manipulate liveness properties because the traditional weak bisimulation equivalence used in CCS cannot. $\pi\tau\omega$-bisimulation is used to reduce a finite state process to a smaller and equivalent one in the compositional adjustment.

It is more feasible for ordinary programmers to adopt the program adjustment approach compared to other methods which synthesize complete programs from (temporal logic) specifications [Manna 84, Emerson 82, Pnueli 90]. The reasons are as follows.

- It is not very difficult for ordinary programmers to produce an FCTI concurrent program, which satisfies at least the functional requirements. A more difficult task is to design and debug the timing of such programs.
- Many bugs are derived from harmful nondeterministic alternatives.
- It is easy for ordinary programmers to specify timing constraints, such as deadlock-free and starvation-free constraints, as compared with implementing them.

1.3 Organization of the Chapter

The remainder of this chapter is organized as follows. Section 2 defines Finite State Processes (FSP) and their equivalence relation and composition operator. Basic and compositional adjustment of FSP is described in Section 3. Section 4 shows a simple and nontrivial example and an experimental result of compositional program adjustment. Finally, Section 5 takes program adjustment in standard programming languages into consideration, followed by related works and summary in Section 6 and 7.

2 Finite State Processes

The basic model for concurrent programs is the finite state process (FSP) defined in Chapter 2, which can specify the finite state transition system with *liveness conditions.* First, we introduce an equivalence relation for FSPs. Then, several operators (composition, relabeling, and reduction) on FSPs are introduced and their properties are shown.

2.1 Equivalence of Finite State Processes

We now introduce *$\pi\tau\omega$-bisimulation equivalence* for FSP which was originally defined for compositional verification in Chapter 5. In this chapter, it is used to reduce an FSP to a smaller and equivalent one in compositional adjustment. $\pi\tau\omega$-bisimulation equivalence is redefined for FSPs as follows.

Definition 38 ($\tau\omega$-divergence)
Let $P = (S, A, L, \delta, \pi, s_0, F)$ be an FSP. $s \in S$ is $\tau\omega$-divergent ($s \uparrow$) if $\forall n > 0.\exists s' \in S.\exists\theta \in A^*.(\mid \theta \mid = n, \hat{\pi}(\theta) = \varepsilon$ and $s' = \delta(s, \theta))$. $\Box$

Definition 39 ($\pi\tau\omega$-bisimulation Equivalence)
Let $P_1 = (S_1, A_1, L_1, \delta_1, \pi_1, s_{01}, F_1)$ and $P_2 = (S_2, A_2, L_2, \delta_2, \pi_2, s_{02}, F_2)$ be FSPs. P_1 and P_2 are $\pi\tau\omega$-bisimulation equivalent ($P_1 \approx_{\pi\tau\omega} P_2$), if there is a binary relation $R \subset S_1 \times S_2$, such that $(s_{01}, s_{02}) \in R$, and $\forall s_1 \in S_1.\forall s_2 \in S_2.(s_1, s_2) \in R \iff$

- *$s_1 \in F_1$ iff $s_2 \in F_2$,*
- *$s_1 \uparrow$ iff $s_2 \uparrow$,*
- *$\forall t_1 \in A_1.\forall s_1' \in S_1.($ if $s_1' = \delta_1(s_1, t_1)$ then*
  $\exists\theta \in A_2^*.\exists s_2' \in S_2.\hat{\pi_1}(t_1) = \hat{\pi_2}(\theta), s_2' = \delta_2(s_2, \theta),$ and $(s_1', s_2') \in R)$,
- *$\forall t_2 \in A_2.\forall s_2' \in S_2.($ if $s_2' = \delta_2(s_2, t_2)$ then*
  $\exists\theta \in A_1^*.\exists s_1' \in S_1.\hat{\pi_2}(t_2) = \hat{\pi_1}(\theta), s_1' = \delta_1(s_1, \theta),$ and $(s_1', s_2') \in R)$.

$\Box$

$\pi\tau\omega$-bisimulation is extended so that it can discriminate designated states and divergence, which cannot be discriminated by weak bisimulation (the weak bisimulation ignores divergences, i.e., τ-loops and τ-circles). The following lemma is derived from these discrimination abilities.

Lemma 9 *If P_1 is complete and $P_1 \approx_{\pi\tau\omega} P_2$, then P_2 is also complete.* $\Box$

Definition 40 (Reduction) *For a given FSP $P = (S, A, L, \delta, \pi, s_0, F)$, a reduction of P, $red(P) = (S_r, A_r, L_r, \delta_r, \pi_r, s_{r0}, F_r)$, is an FSP such that $P \approx_{\pi\tau\omega} red(P)$ and $\mid S_r \mid \leq \mid S \mid$.* $\Box$

The smallest $red(P)$ is constructed effectively by the relational coarsest partitioning algorithm [Paige 87, Kanellakis 90] such that all states of P that are $\pi\tau\omega$-bisimilar to each other are brought together into a single state of $red(P)$.

2.2 Operators on Finite State Processes

Concurrent programs are constructed as a composition of several FSPs that are synchronized with each other. The composition and relabeling operators for FSPs are introduced and their important properties (substitutivity and reflectivity) are shown.

Definition 41 (Composition Operator)
For $P_1 = (S_1, A_1, L_1, \delta_1, \pi_1, s_{10}, F_1)$ and $P_2 = (S_2, A_2, L_2, \delta_2, \pi_2, s_{20}, F_2)$, a composition $P = P_1 \mid P_2$ is defined as follows.
$P = (S_1 \times S_2 \times \{0,1\}^2, (A_1 \cup \{idle\}) \times (A_2 \cup \{idle\}), L_1 \cup L_2, \delta, \pi, (s_{10}, s_{20}, 0, 0), F)$, *where*

- $\delta : (S_1 \times S_2 \times \{0,1\}^2) \times (A_1 \cup \{idle\}) \times (A_2 \cup \{idle\}) \to (S_1 \times S_2 \times \{0,1\}^2) \cup \{\perp\}$ *such that* $\delta((s_1, s_2, f_1, f_2), (a_1, a_2)) =$

$$\begin{cases}
(\delta_1(s_1,a_1), \delta_2(s_2,a_2), f_1', f_2'), \text{ when } \pi_1(a_1) = \pi_2(a_2) \neq \tau, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where } \left\{ \begin{array}{ll} f_i' = 1 & \text{if } \delta_i(s_i,a_i) \in F_i, \\ f_i' = 0 & \text{otherwise}, \end{array} \right\} (\text{ for each } i = 1,2) \\
(\delta_1(s_1,a_1), \delta_2(s_2,a_2), f_1', f_2'), \text{ when } \pi_1(a_1) = \pi_2(a_2) \neq \tau, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where } \left\{ \begin{array}{ll} f_i' = 1 & \text{if } \delta_i(s_i,a_i) \in F_i \vee f_i = 1, \\ f_i' = 0 & \text{otherwise}, \end{array} \right\} (\text{ for each } i = 1,2) \\
(\delta_1(s_1,a_1), s_2, f_1', 0), \text{ when } \pi_1(a_1) \notin (L_1 \cap L_2), a_2 = idle, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where } \left\{ \begin{array}{ll} f_1' = 1 & \text{if } \delta_1(s_1,a_1) \in F_1, \\ f_1' = 0 & \text{otherwise}, \end{array} \right. \\
(\delta_1(s_1,a_1), s_2, f_1', f_2), \text{ when } \pi_1(a_1) \notin (L_1 \cap L_2), a_2 = idle, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where } \left\{ \begin{array}{ll} f_1' = 1 & \text{if } \delta_1(s_1,a_1) \in F_1 \vee f_1 = 1, \\ f_1' = 0 & \text{otherwise}, \end{array} \right. \\
(s_1, \delta_2(s_2,a_2), 0, f_2'), \text{ when } \pi_2(a_2) \notin (L_1 \cap L_2), a_1 = idle, \text{ and } f_1 = f_2 = 1, \\
\quad \text{where } \left\{ \begin{array}{ll} f_2' = 1 & \text{if } \delta_2(s_2,a_2) \in F_2, \\ f_2' = 0 & \text{otherwise}, \end{array} \right. \\
(s_1, \delta_2(s_2,a_2), f_1, f_2'), \text{ when } \pi_2(a_2) \notin (L_1 \cap L_2), a_1 = idle, \text{ and } (f_1 = 0 \vee f_2 = 0), \\
\quad \text{where } \left\{ \begin{array}{ll} f_2' = 1 & \text{if } \delta_2(s_2,a_2) \in F_2 \vee f_2 = 1, \\ f_2' = 0 & \text{otherwise}, \end{array} \right. \\
\perp, \text{ otherwise},
\end{cases}$$

- $\pi : (A_1 \cup \{idle\} \times A_2 \cup \{idle\}) \to L_1 \cup L_2 \cup \{\tau\}$ *such that*

$$\begin{cases}
\pi((a_1,a_2)) = \pi_1(a_1) = \pi_2(a_2) & \text{if } a_1 \in A_1 \text{ and } a_2 \in A_2, \\
\pi((a_1, idle)) = \pi_1(a_1) & \text{if } a_1 \in A_1, \\
\pi((idle, a_2)) = \pi_2(a_2) & \text{if } a_2 \in A_2,
\end{cases}$$

- *and* $F = \{(s_1, s_2, f_1, f_2) \mid s_1 \in S_1, s_2 \in S_2, f_1 = f_2 = 1\}$.

□

We remark that processes are synchronized at actions *with the same labels* in the above process composition. This composition is similar to composition in CCS[Milner 89] except for its treatment of designated nodes. The following relabeling operators are used to relabel actions so that actions which are synchronized in composition have the same labels.

Definition 42 (Relabeling Operator) *For $P = (S, A, L, \delta, \pi, s_0, F)$ and a relabeling function $f : L \to L' \cup \{\tau\}$, $P' = P[f]$ is defined as follows.*

$$P' = (S, A, L', \delta, \pi', s_0, F), \text{ where } \begin{cases} \pi'(a) = f(\pi(a)) & \text{if } \pi(a) \neq \tau, \\ \pi'(a) = \tau & \text{if } \pi(a) = \tau. \end{cases}$$

□

Example 9 (Composition and Relabeling)

- $P_1 = (\{s_0, s_1, s_2\}, \{t_1, t_2, t_3, t_4, t_5\}, \{a_1, b_1, c\}, \delta_1, \pi_1, s_0, \{s_1\})$ *where*
$\delta_1(s_0, t_1) = s_1, \delta_1(s_0, t_2) = s_2, \delta_1(s_1, t_3) = s_2, \delta_1(s_2, t_4) = s_1, \delta_1(s_1, t_5) = s_1, \pi_1(t_1) = a_1, \pi_1(t_2) = b_1, \pi_1(t_3) = b_1, \pi_1(t_4) = a_1, \pi_1(t_5) = c$.

- $P_2 = (\{s_0,s_1,s_2\},\{t_1,t_2,t_3,t_4,t_5\},\{a_2,b_2,d\},\delta_2,\pi_2,s_0,\{s_2\})$ *where* $\delta_2(s_0,t_1)=s_1,\delta_2(s_0,t_2)=s_2,\delta_2(s_1,t_3)=s_2,\delta_2(s_2,t_4)=s_1,\delta_2(s_2,t_5)=s_2,\pi_2(t_1)=a_2,\pi_2(t_2)=b_2,\pi_2(t_3)=b_2,\pi_2(t_4)=a_2,\pi_2(t_5)=d$.
- *relabeling functions:* $f_i(a_i)=a, f_i(b_i)=b$, *and* $f_i(l)=l$ *for other labels* $l\in\{c,d\}$ *(for each i=1,2).*
- $P_1[f_1]\mid P_2[f_2] = (\{s_0,s_1,s_2,s_3,s_4\},\{(t_1,t_1),(t_2,t_2),(t_3,t_3),(t_4,t_4),(t_5,idle),(idle,t_5)\},$ $\{a,b,c,d\},\delta,\pi,s_0,\{s_3,s_4\})$ *where* $\delta(s_0,(t_1,t_1))=s_1,\delta(s_0,(t_2,t_2))=s_2,\delta(s_1,(t_3,t_3))=s_3,\delta(s_1,(t_5,idle))=s_1,\delta(s_2,(t_4,t_4))=s_4,\delta(s_2,(idle,t_5))=s_2,\delta(s_3,(t_4,t_4))=s_1,\delta(s_3,(idle,t_5))=s_2,\delta(s_4,(t_3,t_3))=s_2,\delta(s_4,(t_5,idle))=s_1,\pi((t_1,t_1))=a,\pi((t_2,t_2))=b,\pi((t_3,t_3))=b,\pi((t_4,t_4))=a,\pi((t_5,idle))=c,\pi((idle,t_5))=d$.

(Fig.48) □

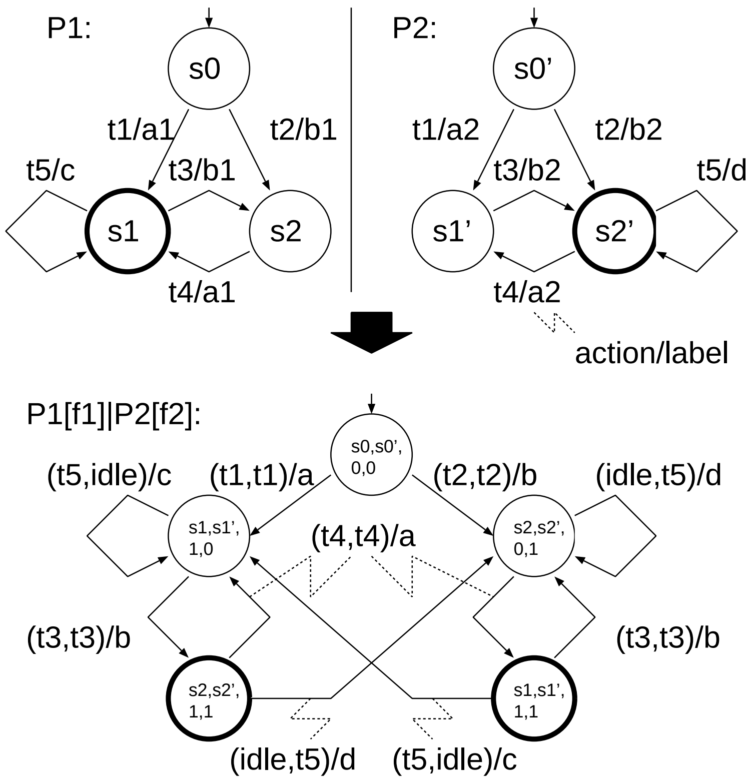


Figure 48. Composition and Relabeling

Definition 43 (Projection) *Let P_1 and P_2 be FSPs. A left projection $L(P_1\mid P_2)\downarrow left$ is defined as $L(P_1\mid P_2)\downarrow left \overset{def}{=} \{\theta_1/\{idle\}\mid \exists\theta\in L(P_1\mid P_2).\theta[i]=(\theta_1[i],\theta_2[i])\}$. Similarly, a right projection $L(P_1\mid P_2)\downarrow right$ is defined. In the same way, projections of L_ω, $L^{fair}_{\Delta_\omega}$, and L_b are defined.* □

Lemma 10 (Reflectivity) *Let P_1 and P_2 be FSPs. If $P=P_1\mid P_2$, then $L_b(P)\downarrow left\subset L_b(P_1)$ and $L_b(P)\downarrow right\subset L_b(P_2)$.* □

Lemma 11 (Substitutivity) *$\pi\tau\omega$-bisimulation equivalence is preserved by composition and relabeling; that is, if $P\approx_{\pi\tau\omega}Q$, then $\forall R.(P\mid R\approx_{\pi\tau\omega}Q\mid R)$, and $\forall f.(P[f]\approx_{\pi\tau\omega}Q[f])$.* □

Reflectivity and substitutivity are used in the basic adjustment and the compositional adjustment, respectively. These adjustments are described in the next section.

3 Program Adjustment

This section proposes program adjustment of FSPs. First, we show that a temporal logic constraint f can be transformed to an equivalent FSP P_f. For an FTCI process P and a temporal logic constraint f, $P \mid P_f$ is a composed process in which P's behaviors against f are disabled by P_f (i.e., safety properties are satisfied). However, $P \mid P_f$ is not necessarily complete (i.e., liveness properties may not be satisfied). Program adjustment means to make $P \mid P_f$ complete by adding arbiter process C (i.e., the adjusted program $= P \mid P_f \mid C$).

3.1 Temporal Logic Constraints

The constraints for concurrent programs (safety properties and liveness properties) are specified by propositional liner time temporal logic (PLTL) [11] . Safety properties include admissible partial ordering of actions (i.e., transition firing), and liveness properties include deadlock and starvation about actions.

Theorem 12 *Given an PLTL formula f under a single event condition, one can build an FSP $P_f = (S, A, L, \delta, \pi, s_0, F)$ such that L corresponds to a set of atomic propositions of f, and $L_b(P_f)$ is exactly the set of behaviors whose label sequences satisfy the formula f.* $\Box$

Proof. This is a FSP version of Lemma 2.

We remark that a label sequence of a satisfiable behavior in P_f corresponds to a model of an PLTL formula.

Example 10 (Temporal Logic Constraints) *Let a label set be $L = \{a_1, a_2\}$.*

- *(1) $\Box\Diamond(a_1 \vee a_2)$: Either a_1 or a_2 must infinitely often occur.*
- *(2) $\Box(a_1 \supset \bigcirc\Box(\neg a_2))$: Whenever a_1 occurs, then a_2 must never occur.*

FSPs which are generated from (1) and (2) are shown in Fig.49.

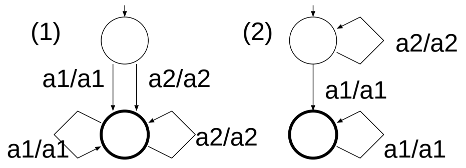


Figure 49. FSPs P_f of Temporal Logic Constraints

In the context of the following program adjustment, we restrict temporal logic formulas so that P_f is **deterministic** with regard to synchronization labels. In this case, some formulas, such as $\Diamond\Box a$, which are translated to nondeterministic one, become not available. These formulas are suitable for verification, but not for adjustment (synthesis) because the arbiter cannot look ahead at future behaviors as indicated by Pnueli and Rosner[Pnueli 89a, Pnueli 89b].

3.2 Basic Adjustment

When temporal logic constraints f can be translated to an FSP P_f, we have to show how to make an FSP $P = P_f \mid P_0$ *complete* for the target FCTI program P_0 by adding an arbiter process C. In other words, basic adjustment is defined as an arbiter synthesis for $P = P_f \mid P_0$ (Fig. 50).

In the following explanation, we assume that the target FSP P has already composed with P_f (i.e., $P = P_f \mid ....$), and do not mention P_f explicitly.

Problem 1 (Basic Adjustment)

[11] Detail definitions are shown in Chapter 2.

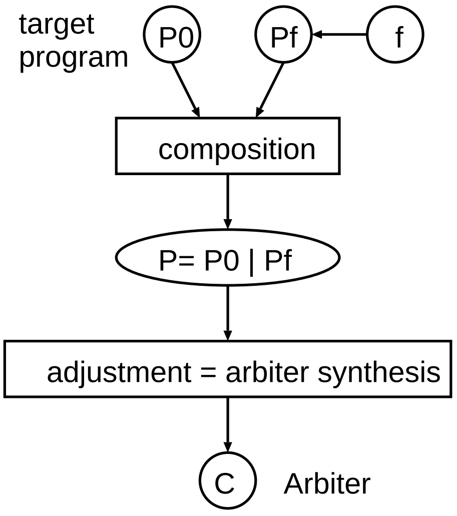


Figure 50. Basic Adjustment

Input: *An FSP $P = (S, A, L, \delta, \pi, s_0, F)$ (We assume $P = P_f \mid ...$).*

Output: *A maximally permissive FSP $C = (S_c, A_c, L_c, \delta_c, \pi_c, s_{0c}, F_c)$ such that $P \mid C$ is complete.*

"C is maximally permissive" means that for every C' if $P \mid C'$ is complete then $L(P \mid C') \subset L(P \mid C)$. □

The arbiter, C, restrains the target FSP P from falling into unsatisfiable states by eliminating harmful observable transitions.

Algorithm 1 (Single Arbiter Synthesis)

(Step 0) $P' := P$.

(Step 1) *Find a set of unsatisfiable states $S_u \subset S'$ in $P' = (S', A', L, \delta', \pi', s'_0, F')$. If there are no unsatisfiable states, go to Step 4.*

(Step 2) *Construct a pseudo-arbiter C' from P' as follows. At first, $\tau - closure$ C_τ is defined as*
$C_\tau(s, a) \overset{def}{=} \{s' \mid \exists\theta.(s' = \delta(s, \theta), \hat{\pi}(\theta) = a)\}$ *for* $\forall s \in S'$ *and* $\forall a \in L \cup \{\varepsilon\}$,
$C_\tau(S_{sub}, a) \overset{def}{=} \bigcup_{s \in S_{sub}} C_\tau(s, a)$ *for* $\forall S_{sub} \subset S'$ *and* $\forall a \in L \cup \{\varepsilon\}$,
then
$C' = (S'_c, A'_c, L, \delta'_c, \pi'_c, C_\tau(s'_0, \varepsilon), S'_c)$, *where* $S'_c = 2^{S'}, A'_c = \{t_a \mid a \in L\} \cup \{t_s \mid s \in S'\}$, *and for* $\forall a \in L$, $\forall s' \in S'_c$,

- $\delta'_c(s', t_a) = C_\tau(s', a) \in S'_c$ *if* $C_\tau(s', a) \cap S_u = \emptyset$,
- $\delta'_c(s', t_a) = \perp$ *if* $C_\tau(s', a) \cap S_u \neq \emptyset$,
- $\delta'_c(s', t_{s'}) = s'$,

and $\pi'_c(t_a) = a$ *and* $\pi'_c(t_{s'}) = \tau$ *for* $\forall a \in L, \forall s' \in S'_c$.

We remark that "$\delta'_c(s', t_a) = \perp$ if $C_\tau(s', a) \cap S_u \neq \emptyset$" means elimination of all behaviors which cannot be distinguished from inevitably unsatisfiable behaviors by a label observer.

(Step 3) $P' := P' \mid C'$, *and return to Step 1.*

(Step 4) *Let the final pseudo-arbiter C', which is generated after applying Step 1 - Step 3 repeatedly, be the arbiter C.*

If C is empty (i.e., all behaviors are eliminated), C is called *unrealizable*; otherwise, C is called *realizable*.

Theorem 13 (Main Theorem) *If an FSP $C = (S_c, A_c, L_c, \delta_c, \pi_c, s_{0c}, F_c)$ is realizable for a given FSP $P = (S, A, L, \delta, \pi, s_0, F)$ in the above algorithm, then $P \mid C$ is complete and C is maximally permissive.* □

Sketch of proof. During Step 1 - Step 3, all inevitably unsatisfiable behaviors are eliminated in the final P'. Therefore, P' is complete. Since the transition function of C' is deterministic about its labels, C' restrains no satisfiable behavior of P. Therefore $P \mid C$ is complete and C is maximally permissive.

Corollary 2

$$L^{fair}_{\Delta\omega}(P \mid C) \downarrow left \subset L_b(P \mid C) \downarrow left \subset L_b(P)$$

□

Proof. This proof is derived from Lemma 1 and Lemma 10 with Theorem 13.

This corollary assures that P, adjusted by C, satisfies its liveness constraints, whenever its behaviors are made by random transitions over states. We remark that an arbiter is effective in case $L^{fair}_{\Delta\omega}(P) \subset L_b(P)$ does not hold (i.e., P has harmful nondeterministic behaviors).

Example 11 (A single arbiter synthesis) *Fig.51 shows a simple single arbiter synthesis. In the target process P, only $\theta = t_3t_6t_7$ is an inevitably unsatisfiable behavior. Since $\{t_3t_6t_7, t_3t_4\}$ is a set of behaviors which cannot be distinguished from θ (i.e. have the same label sequence "ab"), t_4 and t_7 are eliminated. From the remainder, the arbiter C can be constructed.*

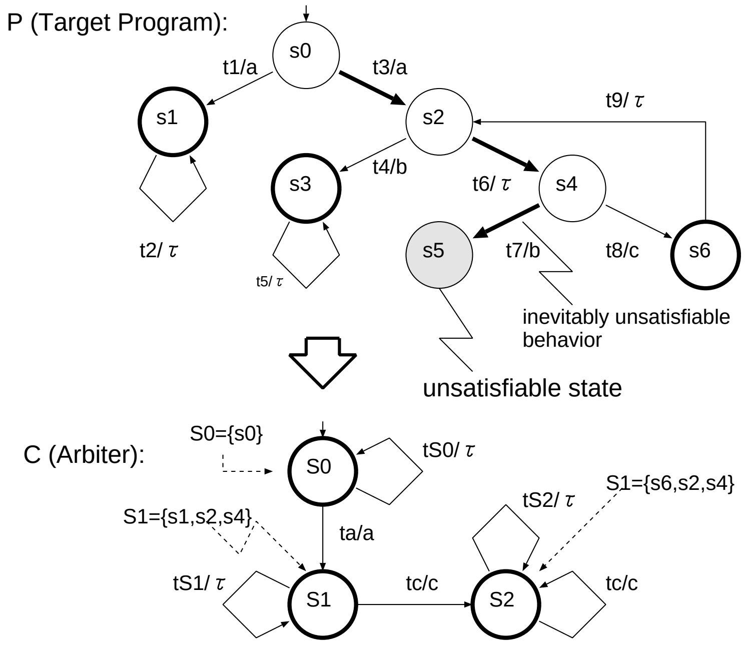


Figure 51. Single Arbiter Synthesis

3.3 Compositional Adjustment

When a target program that is composed hierarchically with many processes becomes very large, the arbiter synthesis may cause the following problems.

1. The synthesis results in a computing cost explosion,
2. A single arbiter is too restrictive to control the whole program precisely.

Therefore, we propose compositional adjustment, in which local arbiters are synthesized in each composition step. The reduction of an FSP can ease the computing cost explosion in each step.

Theorem 14 *If $P_1 \approx_{\pi\tau\omega} P_2$, then C is an arbiter of P_1 iff C is an arbiter of P_2.*
□

Proof. From Lemma 1 and Lemma 11, $C \mid P_1$ is complete iff $C \mid P_2$ is complete.

Corollary 3 *If C is an arbiter of red(P), then C is also an arbiter of P.*
□

Algorithm 2 (Compositional Arbiter Synthesis) *For simplicity, we explain compositional adjustment for the following target program that is constructed by two-level composition (Fig.47(c)). This algorithm can be extended easily to arbitrary target programs.*

- *Target Program:*

$$(P_{11}[h_{11}] \mid P_{12}[h_{12}])[h_1] \mid (P_{21}[h_{21}] \mid P_{22}[h_{22}])[h_2]$$

  where P_{11}, P_{12}, P_{21}, and P_{22} are FSPs, and $h_{11}, h_{12}, h_{21}, h_{22}, h_1$ and h_2 are relabeling functions.

- *Temporal Logic Constraints:*
  f_1,f_2,f_0 are temporal logic constraints for each composition level.

The compositional arbiter synthesis is done in a bottom-up way (Fig. 52).

(Step 1) *Low level arbiters C_1 and C_2 are synthesized for subprocesses $P_{11}[h_{11}] \mid P_{12}[h_{12}] \mid P_{f_1}$ and $P_{21}[h_{21}] \mid P_{22}[h_{22}] \mid P_{f_2}$, respectively. We denote $P_1 \overset{def}{=} (C_1 \mid P_{11}[h_{11}] \mid P_{12}[h_{12}] \mid P_{f_1})[h_1]$ and $P_2 \overset{def}{=} (C_2 \mid P_{21}[h_{21}] \mid P_{22}[h_{22}] \mid P_{f_2})[h_2]$.*

(Step 2) *Reduced subprocesses $red(P_1)$ and $red(P_2)$ are made from P_1 and P_2.*

(Step 3) *A top level arbiter C_0 is synthesized for a target process $red(P_1) \mid red(P_2) \mid P_{f_0}$.*

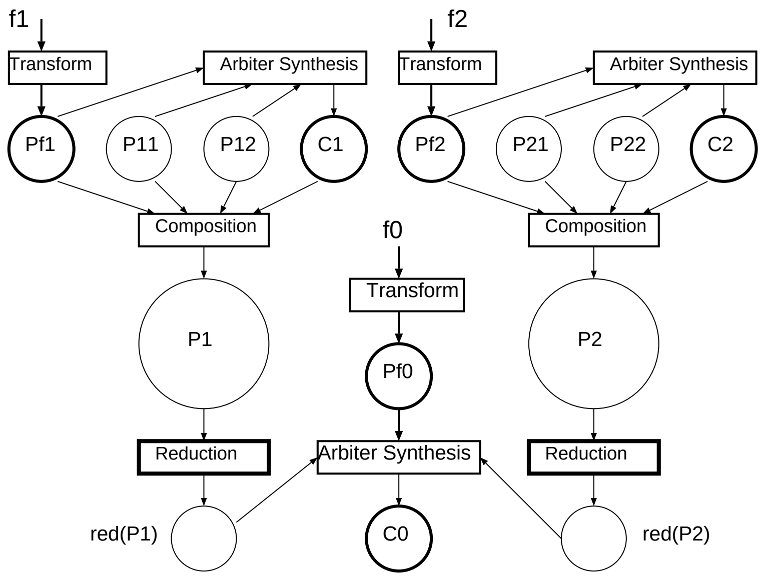


Figure 52. Compositional Arbiter Synthesis

Corollary 2 assures that reduction preserves all information necessary for each local arbiter synthesis. The reduction in each step can cut down the synthesis cost. As the ratio of internal actions in the process increases, so does the effectiveness of the reduction. Note that it is possible to synthesize directly a single arbiter C' for the target programs. However, C' is too restrictive because it has less controllable actions compared with local arbiters, and its synthesis cost is more expensive without reduction. Process reduction by weak bisimulation equivalence has been already proposed and shown its effectiveness in compositional verification by Clarke et. al. [Clarke 89]. However, the reduction preserving liveness properties by $\pi\tau\omega$-bisimulation is our original work.

4 Example and Experimental Result

4.1 Example: The Machine Control Program

In this example we synthesize a single arbiter. The problem may be stated informally as follows. The target program must be designed to control machines which cooperatively process (i.e., etch) printed circuit boards (Fig.53). The coating machine applies resist to boards. The exposure machine exposes boards to the light. The development machine develops boards. The arm machine moves boards from one machine to another. The target program is composed of 6 processes (*Resist*, *Exposure*, *Development*, *Arm*, and *Trans* $\times$ 2) which control corresponding machines. *Trans* represents board transportation. Each process is displayed as a Petri net, shown in Fig.54. With no arbiter, this system is FCTI because it falls into deadlock when an action label sequence of Arm "$get\_r \to put\_e \to get\_r$" occurs. We give the following temporal logic constraints:

$$f = \Box\Diamond(get\_r \lor put\_e \lor get\_e \lor put\_d)$$

which means Arm never falls into deadlock. An arbiter C is synthesized as follows: first, FSPs representing 6 subprocesses are relabeled by relabeling functions $f_r, f_e, f_d, f_a, f_{t1}$, and f_{t2}, and are reduced, and FSP P_f (Fig.55) representing temporal logic constraints f is generated. The target process P (Fig.56) is composed from these FSPs (including P_f). Finally, the arbiter C shown in Fig.57 is synthesized from P, according to Algorithm 1. We can see that the adjusted program "$C \mid P_f \mid Resist[f_r] \mid Exposure[f_e] \mid Development[f_d] \mid Arm[f_a] \mid Trans[f_{t_1}] \mid Trans[f_{t_2}]$" satisfies the above constraints. Figure 58 shows the adjusted program represented by extended Petri net. You can see the target Petri net in Fig. 54 is adjusted by introducing the arbiter C in Fig. 58.

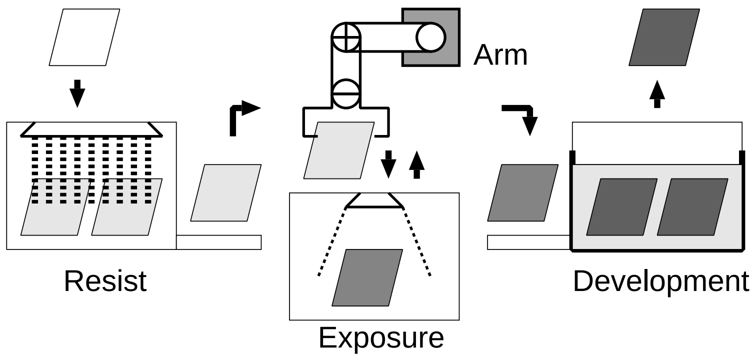


Figure 53. Machine for Processing Printed Circuit Boards

4.2 Experimental Result

We will show how well the compositional method works when it is applied to a middle-scale manufacturing machine control software. This machine is controlled by a concurrent (multi-task) program which consists of 16 element processes (tasks). Table 11 shows the sizes of element processes. The state numbers of each element process may sound small. It attributes to the fact that only synchronization parts of systems are modeled by FSPs.

For this target processes, we give temporal logic constrains by f; prohibition of illegal behaviors of arms and deadlock-freedom for two symmetric process groups ($P_4, P_6, P_7, P_8, P_9, P_{10}$). Two arbiters

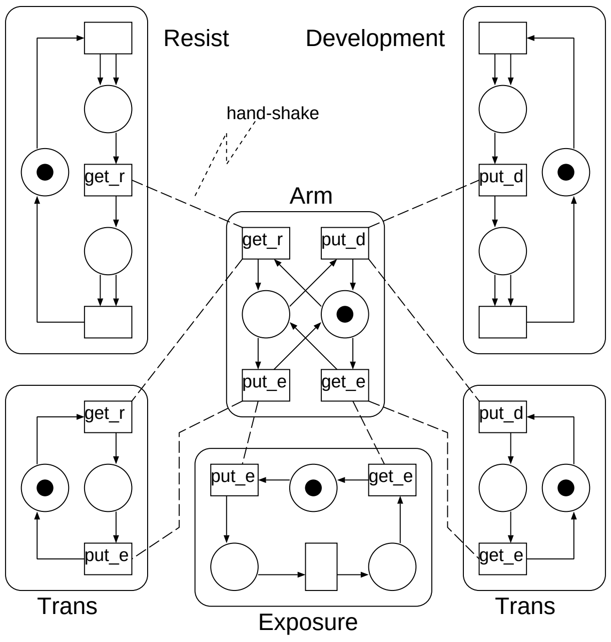


Figure 54. Extended Petri net

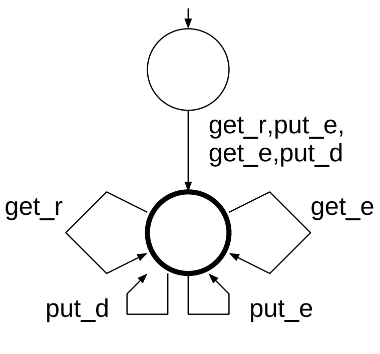


Figure 55. FSP P_f for PLTL formula f

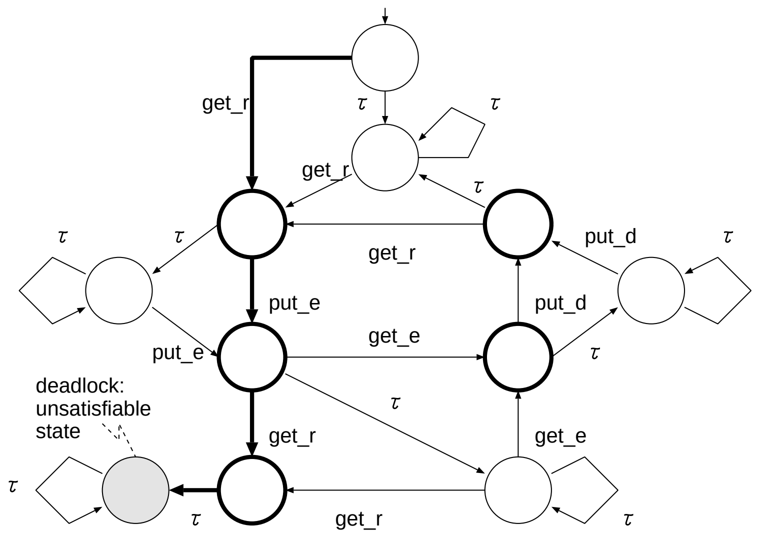


Figure 56. Target Process P(displaying only labels)

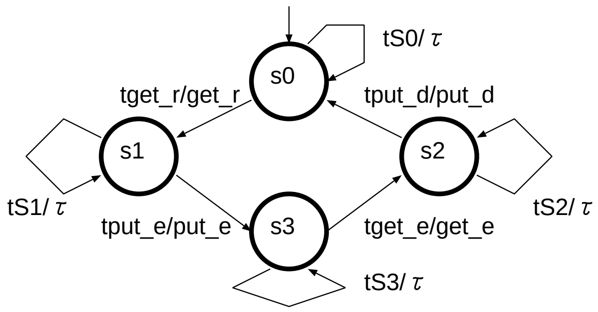


Figure 57. Synthesized Arbiter C

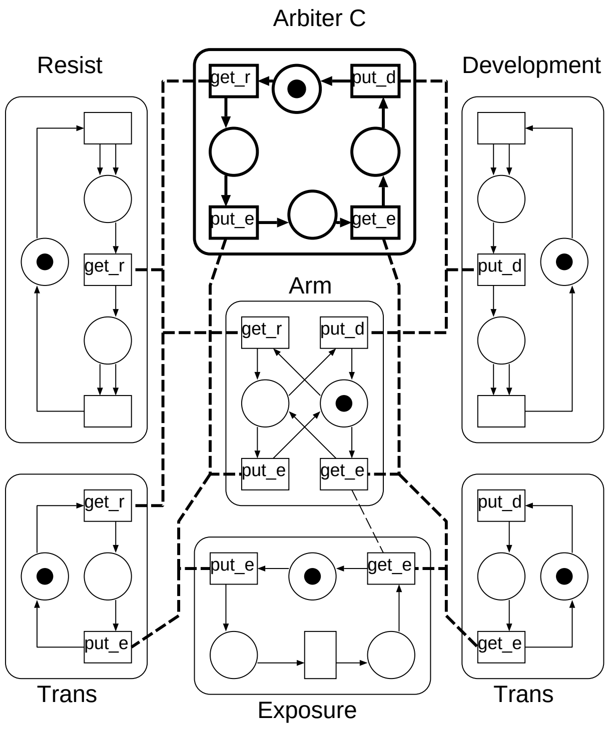


Figure 58. Adjusted Program

were synthesized after the compositional adjustment procedure. Fig. 4.2 shows their whole structure (communication and synchronization among processes).

Table 11. Middle-scale Machine Control Software

Element FSP	*Number of States*
(p1) Distribution Arm	6
(p2) Testing Equipment	3
(p3) 1st Manufacturing Equipment	5
(p4) 2nd Manufacturing Equipment $\times$ 2	3
(p5) 3rd Manufacturing Equipment	5
(p6) Set-up Arm $\times$ 2	6
(p7) Extracting Arm $\times$ 2	6
(p8) 1st door $\times$ 2	2
(p9) 2nd door $\times$ 2	3
(p10)Conveyer $\times$ 2	3

The compositional adjustment procedure to synthesize two arbiters C_{f1} and C_{f2} is shown as follows[12] .

1. $P_{c1} = red(P_4 \mid P_6 \mid P_7 \mid P_8 \mid P_9 \mid P_{10})$ (max_size = 48 states)
2. $P_{c2} = red(P_1 \mid P_2 \mid P_3 \mid P_5)$ (max_size = 85 states)
3. $P_{c3} = red(P_{c1} \mid P_{c2})$ (max_size = 77 states)
4. The first arbiter C_{f1} is synthesized from f and P_{c3} (max_size = 385 states)

[12] Relabeling functions are omitted for simplicity.

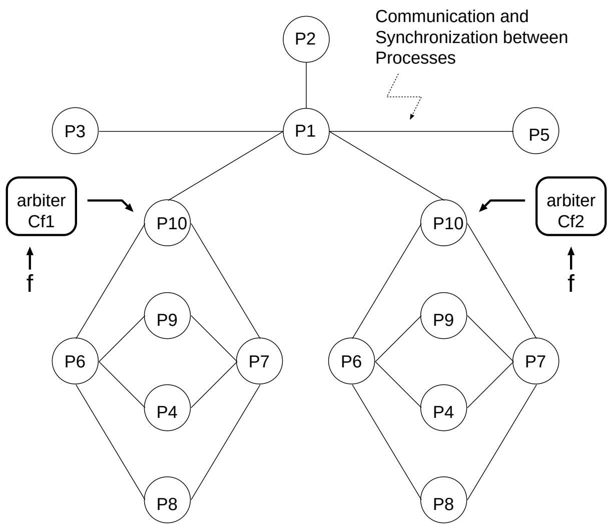


Figure 59. Adjusted Middle-scale Machine Control Software

5. $P_{c4} = red(P_{c3} \mid C_{f1})$ (max_size = 88 states)
6. $P_{c5} = red(P_{c4} \mid P_{c1})$ (max_size = 239 states)
7. The second arbiter C_{f2} is synthesized from f and P_{c5} (max_size = 112 states)

Here, the "max_size" means the maximal number of states which are temporally created during process composition and reduction procedure at each step, and the worst case is $max\_size = 385$. Without the compositional method (i.e, by the basic adjustment), the naive process composition of 16 processes would generate far larger number of states since the max_size is increasing monotonously without reduction. Table 12 shows the maximum number of states in two cases (basic adjustment and compositional adjustment). It says that the compositional method can reduce the maximum size to about 1/150 of the basic adjustment.

In this example, only synchronization parts of the system are modeled by FSPs. If they have a lot of actions which are unrelated to synchronization, which are regarded as τ actions, the process reduction would be more effective.

Table 12. Middle-scale Machine Control Software (Effect of Process Reduction)

Adjustment Type	*Maximum Temporary Size of States*
Basic Adjustment	61096
Compositional Adjustment	385

5 Program Adjustment in Standard Programming Languages

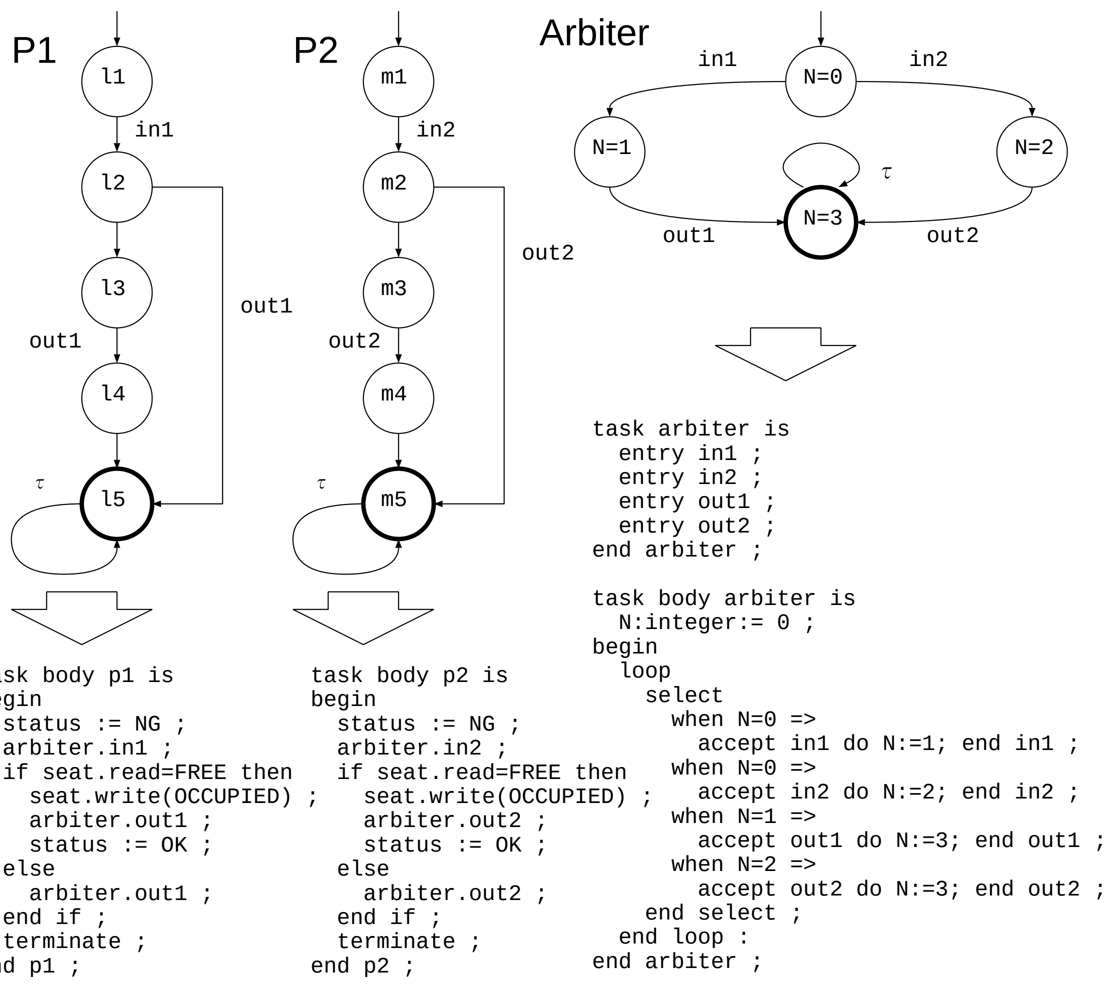
This section considers briefly program adjustment in standard programming languages, instead of FSP. Program adjustment is applicable to concurrent programming languages which have a synchronous (i.e., hand-shake) communication mechanism, like Ada and Occam. For example, Fig. 60 shows a program adjustment example for the Ada program used in the motivation section (Fig. 46). Two FSPs P_1 and P_2 are derived from the original program, then an arbiter is synthesized by the basic adjustment procedure, and finally an adjusted Ada program[13] is derived from FSPs and the arbiter. As you can see in

[13] Some trivial declarations are omitted.

Fig. 60, the arbiter controls the target programs using a *rendezvous* mechanism of Ada to remove harmful nondeterministic behaviors (i.e., θ_3) mentioned in Section 1.

When applying the program adjustment to Ada, we require the following two converters.

- Ada → FSP converter: The Ada program code is divided into basic blocks. Each basic block is assigned to one state of a generated FSP. Control flows between basic blocks are represented as edges between these states. Synchronous communication commands are also represented as edges with synchronization labels. Furthermore, the user can put arbitrary labels on edges which are used to specify temporal logic constraints.
- FSP → Ada converter: A synthesized arbiter represented by a FSP is converted into an Ada task which implements state transitions using *loop* and *select* constructs. Synchronization labels in the arbiter are converted into *accept* commands, and synchronization labels in the target processes are converted into *entry call* commands.



```
task body p1 is
begin
  status := NG ;
  arbiter.in1 ;
  if seat.read=FREE then
    seat.write(OCCUPIED) ;
    arbiter.out1 ;
    status := OK ;
  else
    arbiter.out1 ;
  end if ;
  terminate ;
end p1 ;
```

```
task body p2 is
begin
  status := NG ;
  arbiter.in2 ;
  if seat.read=FREE then
    seat.write(OCCUPIED) ;
    arbiter.out2 ;
    status := OK ;
  else
    arbiter.out2 ;
  end if ;
  terminate ;
end p2 ;
```

```
task arbiter is
  entry in1 ;
  entry in2 ;
  entry out1 ;
  entry out2 ;
end arbiter ;

task body arbiter is
  N:integer:= 0 ;
begin
  loop
    select
      when N=0 =>
        accept in1 do N:=1; end in1 ;
      when N=0 =>
        accept in2 do N:=2; end in2 ;
      when N=1 =>
        accept out1 do N:=3; end out1 ;
      when N=2 =>
        accept out2 do N:=3; end out2 ;
    end select ;
  end loop :
end arbiter ;
```

Figure 60. Program Adjustment in Ada

6 Related Works

Our previous works [Uchihira 87, Uchihira 90a, Uchihira 90b] had proposed program synthesis methods based on temporal logic. However, these methods generated a global state transition graph based on the assumption that all process actions are visible (not internal) and controllable. This assumption is restrictive, and the state transition graph often becomes huge, and its generation is expensive since it cannot be done compositionally. In this chapter, we introduce a CCS-like compositional framework to achieve compositional adjustment utilizing process reduction. Abadi, Lamport, and Wolper [Abadi 89] proposed a compositional program synthesis using the CCS-like compositional framework, where *failure equivalence* is adopted instead of our $\pi\tau\omega$-bisimulation equivalence. However, their approach is a top-down program refinement, which differs from our bottom-up program adjustment approach. From another view, arbiter synthesis can be regarded as a control problem of discrete event systems (*supervisory control*) which are well surveyed by Ramadge and Wonham [Ramadge 89]. However, while these works mainly consider safety properties, they showed no compositional synthesis methods satisfying liveness constraints. The *concurrency control* of database transactions [Bernstein 81] is much related to the program adjustment. Both are intended to remove harmful nondeterminism. The program adjustment can be regarded as the extended concurrency control applied to compositional (hierarchical) concurrent programs.

7 Summary

We have introduced the concept of "program adjustment" into concurrent programming. Program adjustment consists of partially synthesizing programs to remove bugs that are due to harmful nondeterministic behaviors. In the proposed framework, program adjustment is defined as the synthesis of arbiter processes which control target processes with synchronization to satisfy their temporal logic constraints. For compositional adjustment, we have also introduced a new composition and equivalence for finite state processes which can preserve liveness properties, because the traditional CCS framework (composition and equivalence) is not adequate for finite state processes. These techniques are essential to the basic and compositional adjustment.

We also remark that our method is suited for reactive systems which have uncontrollable and unobservable elements in its environment since they can be modeled by τ actions in FSP.

Chapter 7

MENDEL Net: High-Level Petri Net for Reactive and Concurrent Systems

Up to this chapter, Petri net is used as a specification language for reactive and concurrent systems. This chapter focuses on a high-level Petri net as a rather programming language, and proposes a new high-level Petri net, called MENDEL net, which is suited for both specifying and implementing reactive and concurrent systems.

1 Introduction

Although many high-level Petri nets have been proposed, they are not practical enough to describe reactive and concurrent systems in the detail design and implementation phases. They are mainly intended to describe concurrent systems in the modeling phase and are lacking in several important features (e.g. concurrent tasks, task communication/synchronization, I/O interface, task scheduling) for programming reactive and concurrent systems. On the other hand, there are several programming languages based on Petri nets. However, they are deeply depend on its execution environment and not sophisticated as a modeling and specification language.

We propose MENDEL net which is a high-level Petri net extended by incorporating task, task communication/synchronization, I/O interface, and task scheduling in a sophisticated manner. MENDEL nets can bridge the gap between Petri net as specification language and Petri net as programming language.

The remainder of the chapter is organized as follows. First Petri nets as programming languages for reactive and concurrent systems are considered in Section 2. Section 3 introduces MENDEL nets in detail and an example of MENDEL nets is shown in Section 4, followed by related works and a conclusion in Section 5 and Section 6. Finally, a syntax of MENDEL nets is shown in Appendix.

2 Petri Nets as Programming Language

2.1 Programming Language for Reactive and Concurrent Systems

A practical programming language for reactive and concurrent systems requires expressive power for the following items.

- **I/O Interface with Environment**

  I/O interface with an environment (controlled objects) is necessary to define and describe inputs (e.g. sensor information from devices) and output (e.g. control commands to devices) (Fig. 61).

  Concerning inputs, there are two types, *active* and *passive*. An *active* input device generates an interrupt when it has some input to be processed by an *interrupt handler* of the system (controller). As a *passive* input device does not generate interrupt, the system (controller) should read some sensor data by an *input handler* periodically or on demand. Outputs like control commands are sent to devices by an *output handler*. These handlers (input, output, interrupt) are called *device driver* generally.

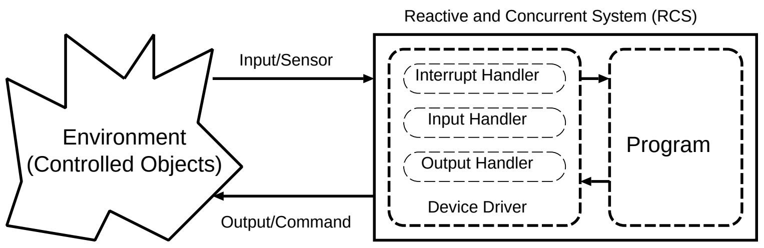


Figure 61. I/O Interface with Environment

- **Concurrent Tasks and Task Communication/Synchronization**

  It is necessary to define concurrent tasks and describe communication and synchronization between these tasks. There are the following mechanisms to realize communication and synchronization [Andrews 83].

  - **communication/synchronization by shared memories**
    semaphore, event flag, etc.
  - **communication/synchronization by message passing**
    mail box (asynchronous message passing), rendezvous (synchronous message passing)

- **Real-Time Task Scheduling**

  Most reactive systems are implemented by multi-tasking on a single processor. They require a real-time task scheduling mechanism to decide which task should be processed by the processor at a given moment. To realize the real-time scheduling, the following mechanisms are necessary and are usually provided by real-time operating systems.

  - Task priority
  - Task dispatching
  - Interrupt handling
  - Real-time management (timer, periodical sampling)

- **Abstraction Mechanism**

  A module (subroutine) and data abstraction (information hiding) are typical abstraction mechanisms in most programming languages. Additionally, the following abstraction mechanisms are effective for reactive and concurrent systems.

  - Abstraction and information hiding for communication and synchronization
  - Abstraction and information hiding for I/O devices

2.2 Extension of Petri Nets as Programming Language

Since standard Petri nets cannot fully satisfy the above requirements for describing reactive and concurrent systems, it is necessary to extend a standard Petri net as follows.

- **I/O Interface with Environment**

  It is necessary to introduce I/O interface with the environment into Petri net explicitly. Concretely, special places and transitions which are linked to device drivers should be defined. For example, Petri net with external inputs and outputs (PNIO) [Ichikawa 85] shows a typical Petri net extension which has I/O Interface.

- **Interface with Other Programming Languages**

  Reactive and concurrent systems may have data and numerical processing parts. It is difficult to extend Petri net to manipulate directly these parts. Therefore, the practical solution is to introduce interfaces with other programming languages for data and numerical processing.

- **Concurrent Tasks and Scheduling Mechanism**

  Although a Petri net has concurrency in itself, the granularity of its concurrency is too small, that is, transition-level. A module-level concurrency (i.e., task, process) should be introduced into Petri nets. Moreover, a scheduling mechanism which controls these tasks should also be expressed within the extended Petri nets.

- **Real-Time Extension**

  To introduce real-time into Petri nets, there are several approaches.

  - Time delays are associated with transitions and/or places. These models include timed Petri nets and stochastic Petri nets [Marsan 86].
  - A global clock is introduced, and then, time stamps are attached to tokens, which are used to describe transition conditions and actions [Bellettini 93].
  - Timers are prepared as built-in subnets, that is, interfaces with the timers are introduced into Petri nets.

- **Individual Tokens and Hierarchical Net Structure**

  By extending tokens to possess individual attributes and values (we call them individual tokens), it is possible to express high-level enabling conditions and actions accompanying transitions. Moreover, it is possible to fold several symmetric nets into a single net by means of individual tokens. On the other hand, hierarchical net structure (e.g., some subnets can be represented by one macro-place or macro-transition) is necessary to design large-scale systems. These individual tokens and net hierarchy are regarded as abstraction mechanisms. Several extended Petri nets have been proposed in which these mechanisms are available. Generally, they are called *high-level Petri nets.*

2.3 Petri-Net-Based Programming Languages

In the field of sequential control, programming languages based on Petri nets are popular, and several languages have been proposed. Some are widely used as languages for programmable logic controller (PLC) in the industry. Individual languages are introduced briefly below.

- SFC

  Traditionally, ladder charts and function blocks are used as programming languages for PLC. However, these languages are structurally flat, and difficult to maintain when the program becomes large. In order to overcome this problem, it is effective to introduce a state-transition-based structure (e.g. Petri net) into ladder charts and function blocks. Sequential Function Chart (SFC) [IEC 1131-3] is a popular programming language for PLC, which is originally based on Petri nets. SFC has interfaces with ladder charts and function blocks. These ladder charts and function blocks are used to describe transition conditions, actions, and I/O interface. SFC has been standardized by International Electrotechnical Commission (IEC).

- High-Level SFC

  Since SFC standardized by IEC is very basic, several extensions have been done by each PLC provider. For example, Instrument Flow Chart (IFC) [Kojima 91] is a high-level SFC for plant control systems (i.e., chemical plants and waterworks and sewage treatment plants), which has the following extensions.

  - Multi-tasking mechanism
  - Domain-specific macro notations

- MFG/PFS

  MFG/PFS [Miyagi 88] is a Mark Flow Graph (MFG) based programming language for control systems for discrete event production systems (DEPS). MFS is a Petri-net-based language which has the following additional features.

  - Specific Tokens: Tokens represent “items” (material, work pieces, etc.) of DEPS.

  - I/O interface: Input and interrupt handlers are represented by specific transitions, and output handlers are represented by specific places.
  - Modularity: A subnet-oriented modularity called “activity” is introduced which means a single production operation of DEPS.

  Production Flow Schema (PFS) is a macro representation of MFG which can support stepwise refinement in designing and programming.

- C-net/SCR

  Control-net (C-net) is a visual programming language for sequential control which is based on a safe coloured Petri net. In C-net, some control program fragments including I/O interface with the environment can be described in each place, which are called *place procedures*. Hence, a net structure of C-net represents a transaction control program where each transaction consists of several place procedures. Station Controller (SCR) [Murata 90] is a programming and executing environment which includes a C-net editor, a C-net interpreter, and an execution monitor. The C-net interpreter supports multi-task processing in which each task is described as a C-net.

- K-NET

  K-NET [Nagao 92] is a programming environment for Flexible Manufacturing Systems (FMS). K-NET adopts a hierarchical high-level Petri net in which enabling conditions of transition and place procedures can be specified by the following user-defined functions and logical I/O functions.

  - **user-defined function:** The user can define functions with C language which are used in enabling conditions and place procedures.
  - **logical I/O function:** Logical I/O functions are also used in enabling conditions and place procedures. In the logical functions, each atomic proposition (logical I/O name) is linked to some physical device.

  The K-NET programming environment consists of editor, simulator, C program generator, monitor, and document generator. Petri net descriptions in K-NET are translated into C programs by the C program generator and compiled and executed by factory computers.

2.4 High-Level Petri Net as Programming Language

Several high-level Petri nets and tools have been proposed, which include Coloured Petri Nets (CPN) and its tool (DESIGN/CPN) [Jensen 92, Jensen 95], Predicate/Transition Nets [Genrich 81] , and Algebraic Petri Nets [Reisig 91] . CPN (DESIGN/CPN) introduces hierarchy constructs into nets to enable a large-scale system description. Although these high-level Petri nets provide sophisticated modeling ability, they cannot be used as a programming (implementation) language for reactive and concurrent systems as they are. Principally, the following extension should be done according to Section 2.2.

- I/O Interface with Environment
- Concurrent Tasks and Scheduling

Since an introduction I/O interface is easy, introduction of concurrent tasks and scheduling mechanism is essential. Hierarchy (module) constructs which these high-level Petri nets provide are subnet-oriented; that is, a part of the net (subnet) is regarded as a hierarchical unit and reduced to one node. This subnet-oriented hierarchy (module) is inadequate to represent concurrent tasks because it does not concern module-level concurrency. Reactive and concurrent systems require task-oriented modules and module composition mechanism as provided in the process theory (e.g., CCS, CSP, ACP). Furthermore, task scheduling cannot be specified explicitly within the framework of these high-level Petri nets.

On the other hand, there are some programming languages (C-net, K-NET) based on high-level Petri nets as mentioned in the previous section. Since they deeply depend on execution environment, they are not sophisticated as specification languages. For example, multi-tasking of SCR is specified as a system configuration outside C-net.

The gap between Petri net as a specification language and Petri net as a programming language is a serious problem in the Petri-net-based software development process. In the next section, we propose MENDEL net to bridge this gap.

3 MENDEL Net

A MENDEL net is a high-level Petri net for both specifying and implementing reactive and concurrent systems. In particular, MENDEL net adopts process-oriented hierarchy and scheduling mechanism by two-level nets, which allows the concurrent tasks, task composition, synchronous/asynchronous communication between them, and task scheduling to be explicitly represented in its model[14] . By this feature, MENDEL net is distinguished from other high-level Petri nets as a specification language (e.g., CPN) and as a programming language (e.g., C-net).

A MENDEL net has the following properties in addition to the standard Petri net.

- four types of places (state element, slot, flag, and port), and port attributes,
- logic program description of transition conditions and actions,
- I/O interface,
- process-oriented net hierarchy,
- two types of communication mechanisms (synchronous and asynchronous) between processes,
- real-time scheduling mechanism by two-level nets, and
- several macro representations.

The above properties not only make MENDEL nets powerful enough to describe most reactive and concurrent systems but also make it possible to automatically retract a skeleton of the MENDEL nets to be used for a net analysis phase (verification and adjustment). MENDEL nets are designed to handle detailed descriptions as well as skeleton-level analysis. The above properties are explained in detail in the following subsections. MENDEL net has both graphical and textual representation. The graphical form is intuitive, but not perfect (i.e., only skeletons are shown). In the following explanation, both representations are used complementarily.

3.1 Place

The places and transitions of standard Petri nets are very general and can have a wide variety of meanings. From the viewpoint of reactive and concurrent programs, places are classified into four types (state elements, slots, flags, and ports) as shown in Fig. 62.

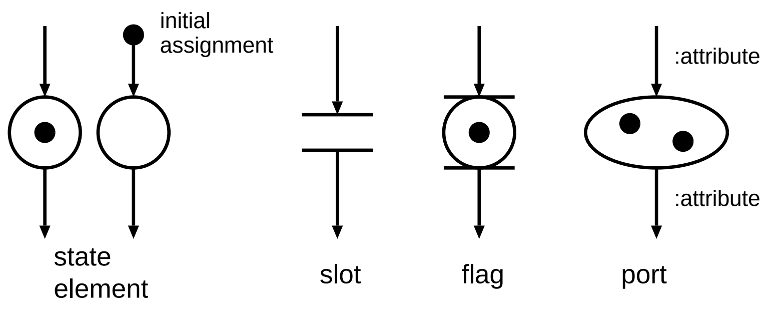


Figure 62. Four Types of Places

- **state element:** A place which represents the local state of a system. This type of place has at most one token (i.e. it is safe). If the place has one token, it means that the system stays in the local state (called "the state element is *active*"). Otherwise, it means that the system does not stay in the local state (called "the state element is *inactive*"). The state elements are graphically represented by circles. Initial assignment of active state elements is represented by arrows starting from bold dots.

[14] MENDEL net uses a term *process* instead of *task*.

- **slot:** A place that represents data and database on a static storage. This type of place has constantly one token. The slots are graphically represented by horizontal bars similar to the data stores in Data Flow Diagram.
- **flag:** A place that represents a Boolean variable on a static storage. If the place has a token, the flag is *true*, otherwise *false*. The flags are often used for process synchronization, that is, used as *event flags*. The flag is graphically represented by combination of two horizontal bars and a circle.
- **port:** A place that represents an infinite buffer necessary for modeling data flow and asynchronous communication. Furthermore, the port may have several *attributes* which are used as indexes of tokens. For example, a token pushed with an attribute *:att* can be popped with *:att*. Each index organizes a FIFO queue. The ports are graphically represented by ellipses.

This classification produces informative structures utilized in Petri net design, analysis, understanding, and code generation. These four types of places are textually declared as follows.

```
states([<state_element_name>,...],[<initial_state_element_name>,...]) ;
slots([<slot_name>(<initial_value>),...]) ;
flags([<flag_name>(<initial_value>),...]) ;
ports([<port_name>(<initial_buffer>),...]) ;
```

Example 12 (Place Declaration)

```
states([s1,s2,s3],[s1]) ;
slots([slot1(10),slot2(ok)]) ;
flags([f1(true),f2(false),f3(true)]) ;
ports([p1([]),p2([1,1,2]),p3([])]) ;
```

3.2 Transition

In MENDEL net, the transition is called a *method* for historical reasons. A method is graphically represented by a rectangle. The method's firing conditions and actions are described in detail by the inscription language based on a logic programming language LPL[15] . Therefore, a MENDEL net is a kind of high-level Petri net, where the individual tokens are represented in logic program terms (*atom*, *integer*, *logical variable*, and *list*), and the conditions and actions are described with *guards* and *actions* of LPL clauses, respectively. The textual form of a method follows:

```
method(<method_name>, <exchange_term>,<input_list>,<output_list> ) :-
                                                  <guard> | <action> ;

<guard> ::= <LPL predicate>, <LPL predicate>, ...
<action> ::= <LPL predicate>, <LPL predicate>, ...
```

The *exchange_term* is used for synchronous communication which is described later. The *input_list* and the *output_list* mean a list of input places and a list of output places of the method, respectively. As the slot and port have individual tokens, they are described in the form: $\langle slot\_name\rangle(\langle term\rangle)$ and $\langle port\_name\rangle(\langle term\rangle)$. Regarding ports, attributes can be used to identify individual tokens in the form: $\langle port\_name\rangle : \langle attribute\rangle(\langle term\rangle)$. In MENDEL net, there is no weight function associated with arcs, and thus, during firing, just one individual token is taken from an input port and just one token is delivered to an output port. The propagation of token information is done by unification of logic program terms of tokens in the same manner as in Prolog. In $\langle guard\rangle$, only LPL predicates which have no side-effect are available.

A firing rule of the MENDEL net is defined as follows.

Enabled Method Search: Search all enabled methods satisfying the following conditions.

- All input state elements of the method are active.

[15] In MENDELS ZONE, the concurrent logic programming language *KL1* [Ueda 90, Chikayama 92] is actually used as a inscription language. However, we can regard the inscription language as Prolog because concurrency of KL1 does not play an important role in MENDEL net. Here, we call it simply Logic Programming Language (LPL).

- All input flags of the method coinside with Boolean value of flags. (`flag` $\leftrightarrow$ *true*, `-flag` $\leftrightarrow$ *false*)
- Each term of the input slot is successfully unified with the term described in the method. When the term of the method is variable, the value of the input slot term is assigned to the variable, and then is referred in the guard and action part.
- Each input port has at least one token. When the port has attributes, there is one token with the same attribute that the method specifies. Each term of the token is successfully unified with the term described in the method. When the term of the method is variable, the value of the input port term is assigned to the variable, and then is referred in the guard and action part.
- The guard condition is true for the assigned variables.

Method Selection: When there are plural enabled methods, select the upper one in the program text.

Method Execution: A selected method is executed as follows.

- Evaluate an *action* part of the method.
- Make all input state elements inactive, and make all output state elements active.
- Make Boolean value of flags coinside with the output flags. (`flag` $\leftrightarrow$ *true*, `-flag` $\leftrightarrow$ *false*)
- Write values evaluated in the *action* part on each output slot.
- Pop up the unified token from every input port, then push one token with the value evaluated in the *action* part into every output port.

A method will be briefly explained by the following example.

Example 13 (Method (Fig. 63))

```
method(move, _, [ready,x(N1),y(M1),type:job(ID)], [busy,x(N2),y(M2),ack(A)]) :-
                    N1>0, M1>0 | N2 := N1+1, M2 := M1+1, A=ok(ID) ;
```

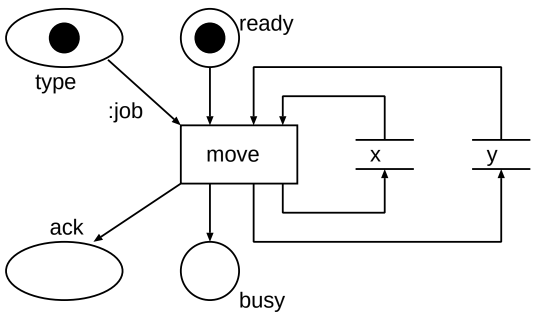


Figure 63. Example of Method

In this method example, the method *move* is enabled if the state element *ready* is active, there is at least one token with an attribute *job* in the port *type* whose term can be unified with a variable ID, the guard ($N1 > 0$ and $M1 > 0$) is satisfied where terms $N1$ and $M1$ are copied from the slots x and y. When the method *move* is executed (fired), the body ($N2 := N1 + 1, M2 := M1 + 1, A = ok(ID)$) is evaluated, the state element *ready* becomes inactive and the state element *busy* becomes active, the evaluated terms $N2$ and $M2$ are written in the slots x and y, and the token is removed from the port *type* and a token whose value is $ok(ID)$ is pushed into the port *ack*.

3.3 I/O Interface

In MENDEL net, I/O interface with the environment is realized by assigning device drivers to distinguished flags, slots, and ports (called I/O flags, I/O slots, and I/O ports). Note that state elements are not available for I/O interfaces. These distinguished places are graphically represented by double lines of the original shapes as shown in Fig. 64. The driver assignment is done using logical addresses *io_tag$⟨logical_address⟩* which indicate physical I/O addresses or program pointers of the driver programs. Actual device drivers are implemented as physical I/O addresses and driver programs which are dependent on the hardware. The ports of Fig. 64 are declared using *io_slots*, *io_flags*, and *io_ports* textually as follows. In this example, *in* and *out* mean I/O mode.

Example 14 (I/O port Example)
I/O slot and port in Fig. 64 are defined textually as follows.

```
io_slots([hand1(io_tag$add1,out)]) ;
io_ports([hand2(io_tag$add2,in)]) ;
```

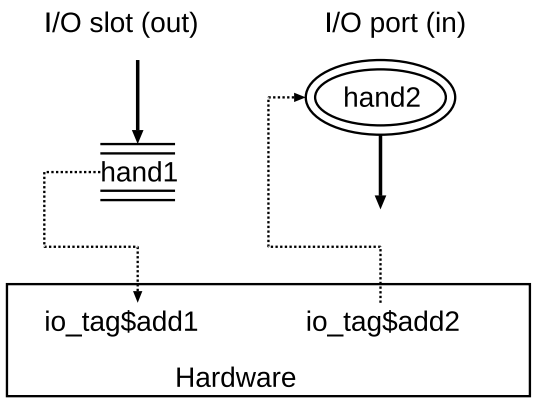


Figure 64. Example of I/O Slot and Port

3.4 Process-Oriented Hierarchy

3.4.1 Overview

In order to enable large-scale system description, it is necessary to introduce module constructs into Petri nets. Many practical high-level Petri nets provide module constructs. However, most of them are classified into *subnet-oriented hierarchy*, that is, a part of net (subnet) is regarded as a hierarchical unit and reduced to one node. This subnet-oriented module is inadequate to represent concurrent processes and process composition of reactive and concurrent systems. Therefore, we propose *process-oriented hierarchy*.

The hierarchical unit of MENDEL nets is a process (i.e., task). A process may consist of several subprocesses hierarchically. While transitions of each process are executed sequentially, the process can run concurrently according to the scheduling mechanism which will be mentioned in the following section. The interaction between hierarchical units is defined as synchronous and asynchronous communication between processes. The process interface is a set of external ports and external methods. A process can push/pop tokens to/from external ports of subprocesses. Since ports are infinite buffers, this interaction realizes asynchronous communication. On the other hand, a process can synchronize its own methods with the external methods of subprocesses (i.e., fire these methods simultaneously only if they are all enabled). Since data exchange is available using the *exchange_term*, this interaction realizes synchronous communication. We emphasize that this process-oriented hierarchy can directly specify a compositional structure of well-researched concurrent process theories, such as CCS, ACP, CSP, and LOTOS. Unlike process theories, MENDEL net allows only fixed composition, and therefore dynamic process creation is unavailable.

Another characteristic feature is *indirect communication* between processes. In MENDEL net, processes can communicate directly only with their parent processes. Communication between processes at the same level is realized indirectly by way of the parent process. This restriction is introduced to secure the benefit of process reuse, since direct communication makes processes strongly interdependent.

3.4.2 Graphical Representation

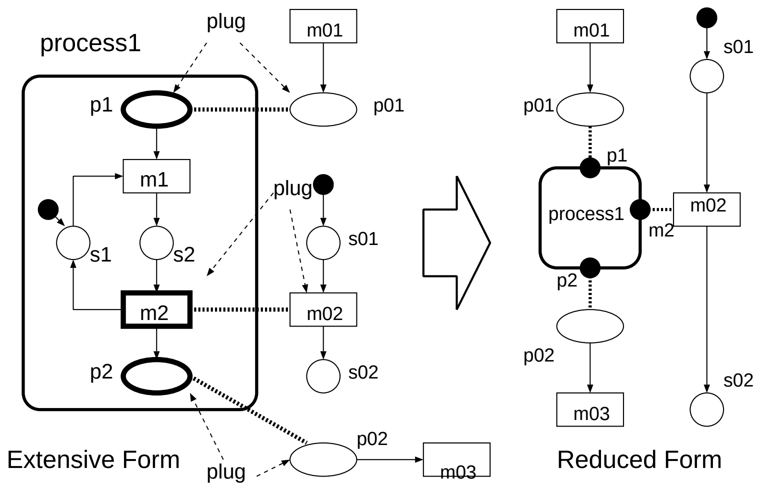


Figure 65. Process-Oriented Hierarchy

Figure 65 illustrates a simple example of a MENDEL net including a process-oriented hierarchy. The external ports and external methods are represented by bold ellipses and bold rectangles, respectively. These external ports and external methods are called *plugs* which mean interfaces with the parent process. The communication between processes is represented by linking dotted lines between places/methods. These dotted lines are semantically interpreted as *transition fusion* and *place fusion* (Fig. 66) introduced in [Christensen 92]. In the transition fusion, fused transitions can exchange data with each other by using ⟨*exchange_term*⟩. From the viewpoint of the parent process, a subprocess is graphically represented by a large circle.

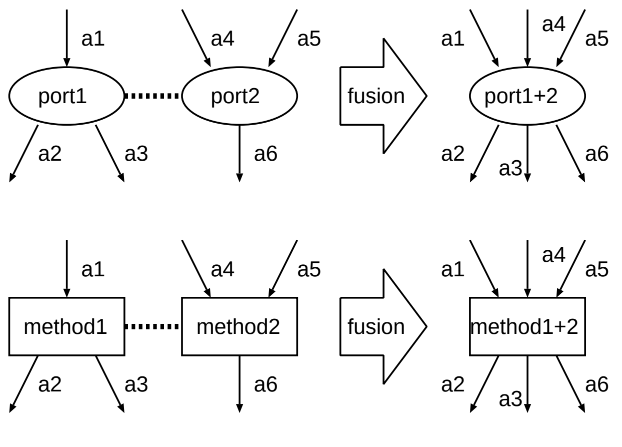


Figure 66. Place and Transition Fusion

The various communication mechanisms can be realized by place and transition fusion as follows.

- **flag fusion:** event flag, semaphore,
- **port fusion:** mail box,
- **method fusion:** rendezvous.

Since transition fusion (method fusion) realizes synchronous communication and place fusion (flag and port fusion) realizes asynchronous communication, MENDEL net can provide two types of communication mechanisms.

3.4.3 Textual Representation

A process textually consists of the following four parts.

- Declaration part (dec:{...})
  declaration of places (state elements, slots, flags, ports)
- Body part (body:{...})
  declaration of subprocesses (`<process_name>(<id>,<method_list>,<place_list>)`)
- Method part (meth:{...})
  definition of methods
- Junk part (junk:{...})
  definition of LPL clauses used in methods

A syntax of a process is defined as follows.

```
process <process_name>( <external_method_list>,<external_place_list>):
  dec:{...} ;
  body:{...} ;
  meth:{...} ;
  junk:{...}
}.
```

Place and transition fusion are specified by argument matching. The following MENDEL net is a textual form of Fig. 65. In this example, ports $p01$ and $p02$ of the parent process "main" are fused with ports $p1$ and $p2$ of the process "process1" respectively, because these arguments are matched in the body part. In the same manner, a method $m02$ of "main" is fused with a method $m2$ of "process1".

```
process main([],[]):{
dec:{
  ports([p01([]),p02([])]) ;
  states([s1,s2],[s1]) ;
} ;
body:{
  process1(pid1,[m02],[p01,p02]) ;
} ;
meth:{
  method(m01,_,[],[p01(v1)]) ;
  method(m02,v2,[s01],[s02]) ;
  method(m03,_,[p02(X)],[]) ;
};
junk:{} ;
}.

process process1([m2],[p1,p2]):
dec:{
  ports([p1(_),p2(_)]) ;
  states([s1,s2],[s1]) ;
} ;
meth:{
method(m1,_,[s1,p1(X)],[s2]) ;
method(m2,X,[s2],[s1,p2(X)]) ;
} ;
junk:{} ;
}.
```

The above example shows communication between a parent process and a child process. Communication between processes at the same level (first cousin processes) is described as follows. In this example, the method $m1$ play the role of communication channel. In the body part, each process is given different process identification (e.g., `pid1`, `pid2`) which makes it possible to create plural instances of the same process.

```
process main([],[]):{
dec:{
  ports([p1([]),p2([])]) ;
} ;
body:{
  process1(pid1,[],[p1]) ;
  process1(pid2,[],[p2]) ;
} ;
meth:{
  method(m1,_,[p1(X)],[p2(X)]) ;
};
junk:{} ;
}.
```

3.5 Process Scheduling Mechanism

In specifying and programming for actual reactive and concurrent systems, a scheduling mechanism plays an important role. Without information about the scheduling mechanism, simulation and analysis of the model may differ from the actual situation and become imperfect in the timing aspect, and usable only for checking the functional aspect. To introduce the scheduling mechanism into the Petri net model, there are three approaches.

- **all-in-one type:** Both an application and a scheduler are described in the same Petri net model.
- **separation type:** An application is described by Petri nets, and a scheduler is described by another model. While the interface with the scheduler may be provided in Petri nets, the scheduling mechanism itself is out of the Petri net model. Most Petri-net-based programming languages (e.g., C-net/SCR) are of this type.
- **two-level type:** An application and a scheduler are described in two Petri nets of different levels; *base-level net* and *meta-level net*. A base-level net describes an application and a meta-level net describes a scheduler.

Since a Petri net is not only used to specify concurrent programs but also suitable for describing schedulers [Vallejo 94], we adopt the two-level net approach for MENDEL net (Fig. 67). Merits of the two-level net include model consistency as compared with the separation type and easiness of description and maintenance as compared with the all-in-one type.

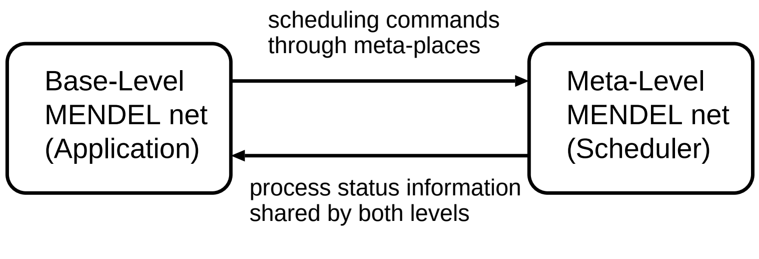


Figure 67. Two-Level MENDEL Nets

To specify interaction between base-level net and meta-level net, we introduce the notions of *meta-place* and *token sharing.*

- **meta-place:** A base-level net has special places, called meta-places, which are shared with a meta-level net by place fusion, and used to communicate with a meta-level net. A token that is put into a meta-place in the base-level net is taken out and used in a meta-level net. These tokens represent scheduling commands in MENDEL nets.

- **token sharing:** Some tokens in a base-level net can be shared with a meta-level net. Concretely, a meta-level net can change information of shared tokens which are referred to in a based-level net. Shared tokens are used to change and refer to the status of processes from both a base-level net and a meta-level net.

In the following sections, the process scheduling mechanism using two-level nets, token sharing, and meta-place is described in detail.

3.5.1 Base-Level MENDEL Net

Since a base-level MENDEL net has already been described in the previous sections (3.1 – 3.4), we explain several scheduling commands (tokens) sent from the application (base-level net) to the scheduler (meta-level net) via meta-places. Primitive scheduling commands include `sta_prc`, `ter_prc`, and `sus_prc` which are also used in the real-time operating system μ-ITRON [Fukuoka 91] are briefly explained as follows.

- *sta_prc(Proc,Prio)*: start the process at a processor *Proc* with a priority *Prio*.
- *ter_prc*: terminate the process.
- *sus_prc*: suspend the process.

One meta-place is established corresponding to each process. In other words, each process has one meta-place which accepts scheduling commands from other processes or from the process itself. We call the meta-place *process place*. Graphically we use a large circle representing a process for its process place (Fig. 68).

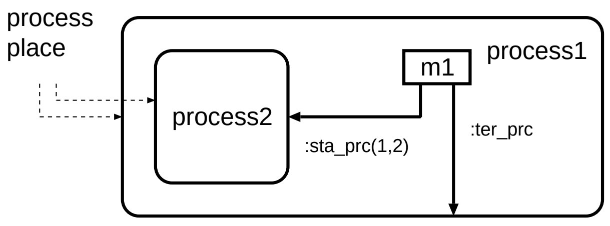


Figure 68. Process Place and Scheduling Commands

A process place is textually identified as ⟨*process_id*⟩. When sending commands to itself, *self* is used instead of ⟨*process_id*⟩. For example, a MENDEL net of Fig. 68 is textually described as follows. In this example, *p*1 represents ⟨*process_id*⟩.

```
process process1([],[]):{
  dec:{}
  body:{
    process2(p1,[],[]) ;
  } ;
  meth:{
    method(m1,_,[],[p1:sta_prc(1,2),self:ter_prc]) ;
  } ;
  junk:{}
}.
```

3.5.2 Meta-Level MENDEL Net

A scheduler is described as a meta-level net. A meta-level net manipulates tokens representing process status, called *process status token*, which are initially created by the *sta_prc* command. When the *sta_prc* command is sent to a process place in the base-level net, all initial state elements of the corresponding process are marked with tokens which have a *process status information PSI*. At the same time, the corresponding process place in the meta-level net has a process status token which has the same PSI. It is token sharing; tokens in initial state elements in the base-level net and a token in a process place in the meta-level net share the same PSI. While usual individual tokens have *values*, these tokens have *references* to the shared PSI (Fig. 69).

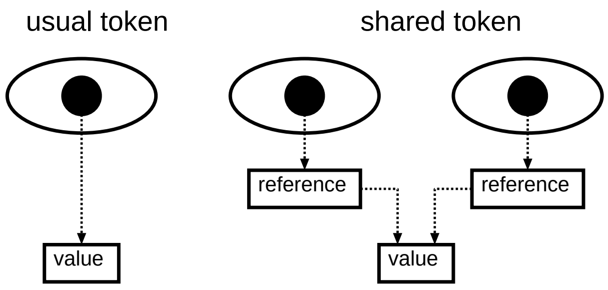


Figure 69. Shared Token

There are the following process statuses in PSI.

- `PSI.meta`: process status changed by a meta-level net
  e.g., $PSI.meta \in \{active, ready, wait, suspended, wait\_suspended, dead\}$
- `PSI.base`: process status changed by a base-level net
  e.g., $PSI.meta \in \{enabled, disabled\}$

A process scheduler (meta-level net) can change process status (`PSI.meta`), which indicates a place with the corresponding process token. In the base-level net, only active tokens (i.e., tokens PSI of which is *active*) are available at method firing. As dead tokens can never be changed to active unless the process is restated again, they can be removed from the base-level net. On the other hand, a base-level net can change process status, *enabled* and *disabled*. When there are enabled methods regardless of `PSI.meta`, a process status becomes *enabled*, otherwise it becomes *disabled*. This status information (*enabled* or *disabled*) is used for process dispatching in the scheduler.

For example, a simple scheduler is described in Fig. 70 which is based on subset of μ-ITRON for a single processor. A place *process_if* is an interface with a base-level net which forms place fusion with process places. Functional inscriptions of transitions are described by LPL as follows.

- **t1 (dispatch):** The scheduler selects a process which is in the *ready* place and has the highest priority and moves it to the *active* place.
- **t2 (preempt):** When a process has just become in the *ready* place and has higher priority than the process in the *active* place, the scheduler exchanges status of these processes.
- **t3 (wait):** When the active process has no enabled transitions in the base-level net (i.e., $PSI.base = disabled$), it is moved from the *active* place to the *wait* place.
- **t4 (ready):** When the waiting condition of the *wait* process is satisfied in the base-level net (i.e., $PSI.base = enabled$), it is moved from the *wait* place to the *ready* place.
- **t5 – t18:** omitted.

3.6 Timer

Concerning real-time extension of Petri nets, we adopt timers prepared as a built-in subnet,

```
sys$timer(<tid>,[],[op,up]).
```

Here, a setting/resetting port (`op(set)`,`op(reset)`) and a referring flag (`up`) mean the interface of the timer.

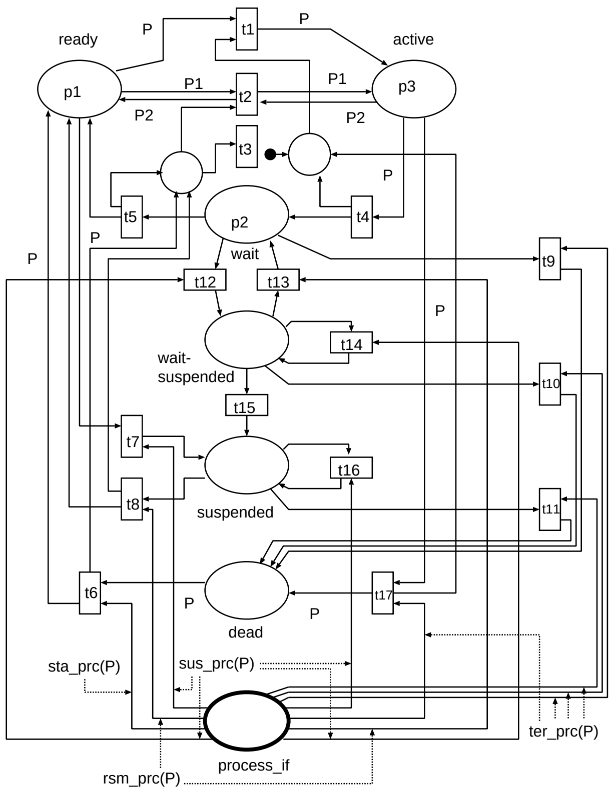


Figure 70. Simple Scheduler Example (μ-ITRON)

```
process process([],[]){
dec:{
  states([s1,s2],[s1]) ;
  ports([op([])]) ;
  flags([up]) ;
};
body:{
  sys$timer(t1,[],[op,up]) ;
} ;
meth:{
  method(m1,_,[s1],[s2,op(set)]) ;
  method(m2,_,[s2,up],[s1,op(reset)]) ;
} ;
junk:{}
}.
```

3.7 Macro Notations

Generally speaking, actual descriptions by Petri nets are often troublesome. Macro notations are very useful in practice. Here, one of the macro notations, array representation, is introduced.

Array Representation Occasionally, a reactive and concurrent system contains several subprocesses having the same structure. For example, a lift system may have several request buttons corresponding to every floor. It is tedious to individually write all button processes. To overcome this problem, coloured Petri nets represent these subprocesses as separate coloured tokens on a single net structure. However, this approach is not suitable for the process-oriented hierarchy, because a process is a hierarchical unit and tokens should not be a process. Therefore, a MENDEL net provides an alternative: an array to represent several subprocesses with the same structure, in the same manner as the CSP-based concurrent programming language Occam. Each arrayed process can be explicitly treated as a separate computing unit that can run on a separate CPU. Figure 71 shows a graphical array representation of N identical processes, whose textual form follows.

$$process\{I\}(m\{I\})(in\{I\})(out\{I\}) * I : \{1..N\}$$

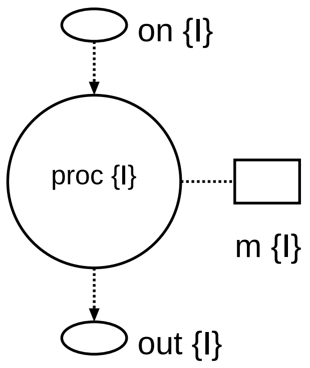


Figure 71. Graphical Representation of Array

4 Simple Example

A simple example of a MENDEL net (base-level net) is shown in Fig. 72, which describes a part of a control program of a telephone terminal. This MENDEL net specifies how the phone terminal becomes connected. The MENDEL net consists of five processes (main, phone, calling, hook, and ring), where process *main* activates other processes by sending *sta_prc* commands to process places. In these processes *calling* and *hook*, I/O ports *call_driver* and *up_driver* are defined which detect input events from the environment, calling and taking up the receiver, respectively. After detecting calling, a process *phone* receives a token from a port *call* of the process *calling* and makes a flag *bell* of a process *ring* active. In succession, the process *ring* sets an output slot *bell_driver on* and *off* to ring a bell intermittently using a built-in subnet *sys$timer*. Finally, after detecting taking up the receiver, the process *phone* makes the flag *bell* inactive, then the process *ring* stops ringing the bell.

```
/* main */
process main([],[]):{
  dec:{
    states([start],[start]) ;
    ports([call([]),up([])]) ;
    flags([bell(false)]) ;
  };
  body:{
    phone(p1,[],[call,up,bell]) ;
    calling(p2,[],[call]) ;
    hook(p3,[],[up]) ;
    ring(p4,[],[bell]) ;
  };
  meth:{
    method(m1,[start],[p1:sta_prc(1,4),p2:sta_prc(1,2),p3:sta_prc(1,1),p4:sta_prc(1,3)) ;
  };
  junk:{};
}.


/* phone */
process phone([],[call,up,bell]):{
  dec:{
```

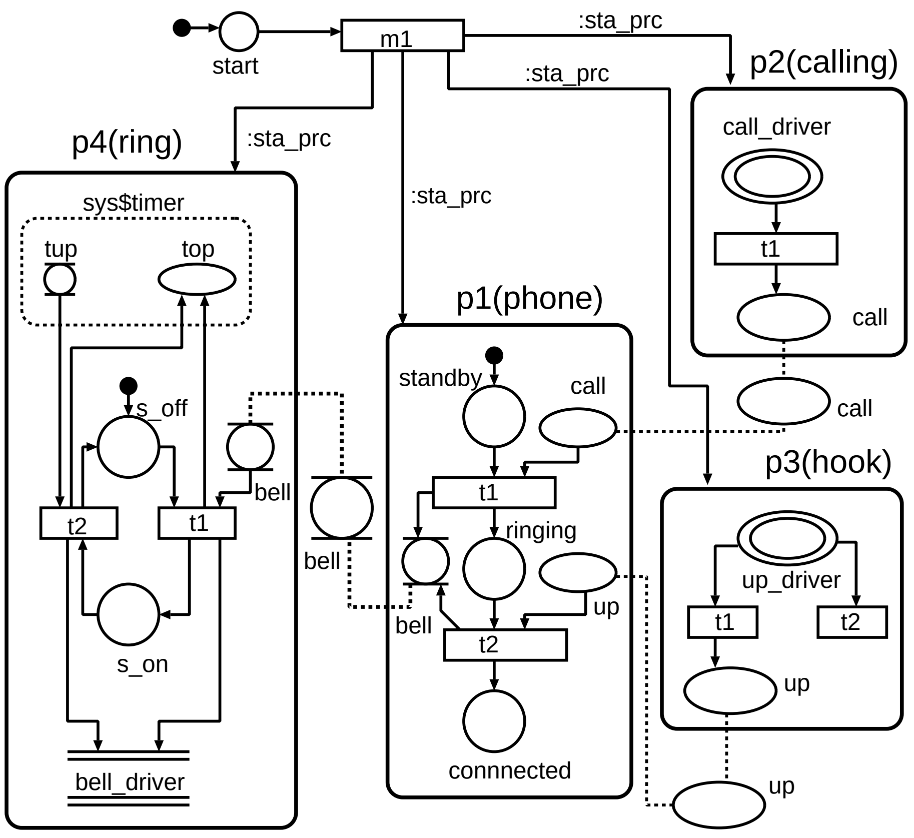


Figure 72. Example (Telephone Terminal)

```
    ports([call(_),up(_)]) ;
    flags([bell(_)]) ;
    states([standby,ringing,connected],[standby]) ;
  };
  body:{};
  meth:{
    method(t1,_,[standby,call(on)],[ringing,bell]) ;
    method(t2,_,[ringing,up(on)],[connected,-bell]) ;
  };
  junk:{};
}.

/* calling */
process calling([],[call]):{
  dec:{
    io_ports([call_driver(io_tag$call,in)]) ;
    ports([call(_)]) ;
  };
  body:{};
  meth:{
    method(t1,_,[call_driver(_)],[call(on)]) ;
  };
  junk:{};
}.

/* hook */
process hook([],[up]):{
  dec:{
    io_ports([up_driver(io_tag$up,in)]) ;
    ports([up(_)]) ;
  };
  body:{};
  meth:{
    method(t1,_,[up_driver(up)],[up(on)]) ;
    method(t2,_,[up_driver(_)],[]) ;
  };
  junk:{};
}.

/* ring */
process ring([],[bell]):{
  dec:{
    io_ports([bell_driver(io_tag$bell,out)]) ;
    ports([top([])]) ;
    flags([bell(_),tup(false)]) ;
    states([s_off,s_on][s_off]) ;
  };
  body:{
    sys$timer(tim,[],[top,tup]) ;
  };
  meth:{
    method(t1,_,[s_off,bell],[s_on,top(set),bell_driver(on)]) ;
    method(t2,_,[s_on,up],[s_off,bell_driver(off),top(reset)]) ;
  };
  junk:{};
}.
```

5 Related Works: Petri Nets and Hierarchy

Several hierarchies in Coloured Petri Net (CPN) were proposed in [Huber 90]. They introduced the notion of *pages* (i.e., net modules), *substitution transitions* and *substitution places*, where a substitution transition/place is a macro node and represents a subpage which contains the detail of how it actually performs the activity. It means *subnet-oriented net hierarchy*. Only the substitution transition is adopted in DESIGN/CPN and this version is usually called Hierarchical Coloured Petri Net (HCPN). Modular Coloured Petri Net (MCPN) [Christensen 92] extends HCPN by incorporating both place and transition fusion. A more flexible form of transition fusion was also proposed by extending CPN with synchronization channels [Christensen 94]. MCPN and CPN with synchronization channels are intended to represent a *process-oriented net hierarchy*, which is similar to the hierarchy of MENDEL nets although they have been proposed apart from MENDEL net. Furthermore, Object Petri Net (OPN) [Lakos 95] has not only a process-oriented and subnet-oriented net hierarchy but also a *token-oriented net hierarchy*, where tokens are allowed to be subnets encapsulating their own activity [16] . Other approaches of process-oriented net hierarchy include *OBJSA net* [Battiston 88] and *Protob* [Bruno 95]. OBJSA nets combine nets with algebraic specification techniques and support process composition by transition fusion. Process composition by transition fusion is advantageous for compositional analysis techniques. OBJSA net is similar to LOTOS which combines CCS with algebraic specification techniques, where label synchronization of LOTOS is the same as transition fusion, and *G-LOTOS* [Bolognesi 89, Lee 91] provides a graphical representation with a process-oriented hierarchy of LOTOS. Protob is a high-level Petri net in which systems can be structured according to the principles of object orientation. An object model is a system's component and its behavior is given by a net. Objects are enabled to communicate with other objects by interface places. The structure of objects in Protob is a tree; a compound object contains other objects. This compositional structure is very similar to MENDEL net. However, Protob is, if anything, a specification language and objects in Protob do not directly correspond to concurrent processes. Compared with these high-level nets with process-oriented net hierarchy, MENDEL nets are strongly process-oriented because process scheduling mechanism and I/O devices can be explicitly specified within the MENDEL net framework. In other words, MENDEL nets are intended to be a programming language rather than a specification language.

6 Summary

This chapter considered a Petri net as a programming language for reactive and concurrent systems. Although several Petri-net-based programming languages for reactive and concurrent systems have been proposed, they are short of abstraction mechanisms and there is a large gap between Petri net as a specification language and Petri net as an implementation language. On the other hand, high-level Petri nets are lacking in several important features (e.g. concurrent tasks, task communication/synchronization, I/O interface, and task scheduling) for programming reactive and concurrent systems. Therefore we proposed a high-level Petri net, MENDEL net, for reactive and concurrent systems to bridge the gap between a specification language and an implementation language.

A MENDEL net features the following properties in addition to standard Petri nets.

- I/O interface,
- process-oriented net hierarchy, and
- task scheduling mechanism modeled by two-level net approach.

MENDEL nets are used as a specification and programming language in the programming environment MENDELS ZONE.

Syntax of MENDEL Net Textual Representation

```
<program> ::= <process> | <program>

<process> ::=
  'process' <process_name> '(' <external_method_list> ',' <external_place_list> ')'
        ':' '{'
                <declaration_part> ';'
                <body_part> ';'
                <method_part> ';'
                <junk_part> ';'
        '}' '.'
```

[16] There have been several proposals for combining object-orientation with Petri nets. They are classified into *object inside nets* and *Petri nets inside objects*; in the former tokens are objects (e.g., OPN), in the latter Petri nets are used for specifying the behavior of individual objects (e.g., HOOD nets [Giovanni91]). In the latter approach, hierarchy is not necessarily provided by Petri nets.

```
<declaration_part> ::=
  'dec' ':' '{'
    <declaration_list>
  '}'

<declaration_list> ::=  <declaration> ';' | <declaration_list>

<declaration> ::=
  <state_declaration> |
  <slot_declaration> |
  <flag_declaration> |
  <port_declaration>

<state_declaration> ::=
                'states' '(' <state_list> ',' <initial_state_list> ')'

<state_list> ::=
       '[' { <state_element_name> ',' }* <state_element_name> ']' | '[' ']'

<initial_state_list> ::=
       '[' { <state_element_name> ',' }* <state_element_name> ']' | '[' ']'

<slot_declaration> ::=
        'slots' '(' <slot_list> ')' |
        'io_slots' '(' <io_slot_list> ')'

<slot_list> ::= '[' { <slot_name> '(' <initial_value> ')' ',' }*
                      <slot_name> '(' <initial_value> ')' ']'
              | '[' ']'

<io_slot_list> ::= '[' {<io_slot> ',' }*  <io_slot> ']' | '[' ']'

<io_slot> ::= <slot_name> '(' 'io_tag' '$' <logical_address>, <io_mode> ')'

<flag_declaration> ::=
                'flags' '(' <flag_list> ')' |
                'io_flags' '(' <io_flag_list> ')'

<flag_list> ::= '[' { <flag_name> '(' <initial_value> ')' ',' }*
                      <flag_name> '(' <initial_value> ')' ']'
              | '[' ']'

<port_declaration> ::=
                'ports' '(' <port_list> ')' |
                'io_ports' '(' <io_port_list> ')'

<port_list> ::= '[' { <port_name> '(' <initial_value> ')' ',' }*
                      <port_name> '(' <initial_value> ')' ']'
              | '[' ']'

<body_part> ::= 'body' ':' '{'   { <process_call> ; }*   '}'

<process_call> ::= <process_name> '('
                         <process_id> ','
                         <external_method_list> ','
                         <external_place_list> ')'

<method_part> ::= 'meth' ':' '{'   { <method> ; }*   '}'

<method> ::= 'method(' <method_name> ',' <exchange_term> ','
                       <input_place_list> ',' <output_place_list> ')' ':-'
                       <guard> '|' <action>

<input_place_list> ::=  '[' { <place> ',' }* <place> ']' | '[' ']'

<output_place_list> ::= '[' { <place> ',' }* <place> ']' | '[' ']'

<place> ::= <state_element> | <flag> | <slot> | <port>

<state_element> ::= <state_element_name>
```

```
<flag> ::= <flag_name>

<slot> ::= <slot_name> '(' <LPL_term> ')'

<port> ::= <port_name> '(' { <LPL_term> | <attribute> ':' <LPL_term> } ')'

<guard> ::= { <LPL_predicate> ',' }* <LPL_predicate>

<action> ::= { <LPL_predicate> ',' }* <LPL_predicate>

<junk_part> ::= 'junk' ':' '{'   { <LPL_clause> ; }*   '}'
```

Chapter 8

Petri-Net-Oriented Design Methodology

This chapter proposes a design methodology for MENDEL nets. Although many Petri net tools have been proposed, most tools support only drawing, simulation, and analysis of Petri nets; few tools support the design methodology for Petri nets. While Petri nets are good final design documents easy to understand, analyzable, and executable, it is often difficult to write Petri nets directly in an earlier design phase when the system structure is obscure. A proposed design method makes a designer to construct MENDEL nets stepwise and systematically using *causality matrices* and *temporal logic.*

1 Petri-Net-Oriented Design Methodology

As mentioned in Chapter 1, Petri nets are often inadequate in an early design phase when some parts of the system structure may be obscure. Particularly, while the structures of local processes, called elementary processes, are usually tangible and able to be easily designed, the structures of a process to coordinate elementary processes, called coordinator process, are often obscure at first in a reactive and concurrent systems. Since a Petri net is a formal language and does not permit vagueness, it is difficult to design systems from beginning to end using only Petri net.

We focus on control systems which are typical instances of reactive and concurrent systems, and then propose a design method for them utilizing MENDEL net and additional complementary formalisms (causality matrix and temporal logic). The *causality matrix* represents causality relations among system elements. It allows vagueness (abstract level description), so a designer can stepwise refine the causality matrix from an abstract level to a concrete level. Furthermore, draft Petri nets which are synthesized from the causality matrix are validated using temporal logic. By doing these stepwise refinements, the designer can construct a correct MENDEL net systematically.

The proposed design method consists of the following phases.

1. Initially elementary processes (i.e., tangible or concrete processes) which have interfaces with controlled objects (environment) and manipulate them are designed by MENDEL nets.
2. Coordinators (i.e., conceptual processes) are created to coordinate elementary processes and elementary processes are interconnected with them by communication channels.
3. Coordinators are designed by stepwise refinement using the causality matrix, and MENDEL nets are finally produced which represent these coordinators.
4. A target system which consists of elementary processes and coordinators are described by MENDEL nets and are validated by simulation, verification, and adjustment.

The remainder of this chapter is organized as follows. Section 2 shows a typical software architecture of control systems. The causality matrix is described in Section 3. A detail procedure of the proposed design method is given in Section 4. Section 5 describes a MENDEL net design example for a lift system, followed by related works and a summary in Section 6 and Section 7.

2 Software Architecture of Control Systems

First we show a typical model of control software (Fig. 73). In this model, control software consists of elementary processes and coordinator processes.

Elementary process: This process corresponds to some controlled object in the problem domain and manages I/O devices to control them. For example, in a plant control system, valves pumps, and motors are controlled objects.

Coordinator process: This process corresponds to a set of functions required for controlling controlled objects. This process coordinates elementary processes to accomplish these functions. The coordinator may be consists of a set coordinator processes.

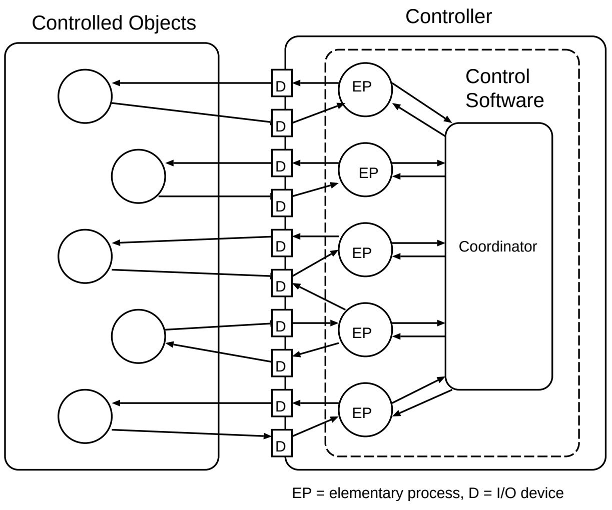
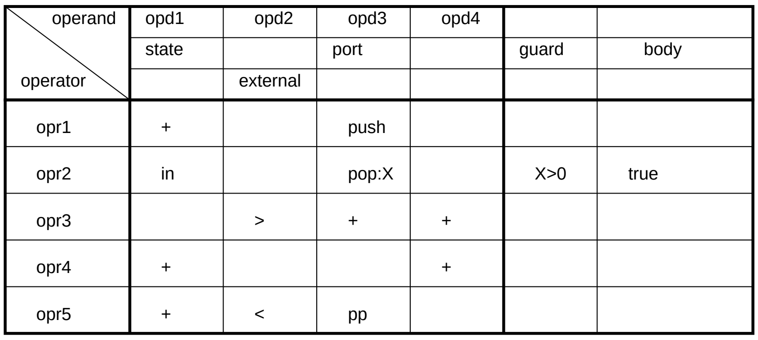


Figure 73. Model of Control Software

3 Causality Matrix

The causality matrix, C=[c_{ij}], is a type of an extended incidence matrix of a Petri net (Fig. 74), where each entry, c_{ij}, represents a causality relation between an i-th operator (i.e., method) and a j-th operand (i.e., state element, flag, slot, port, or external method). One of unique features of the causality matrix is its wide-spectrum property that the causality relation of each entry can range from an abstract level to a concrete level (we call this a wide-spectrum causality relation). Table 13 shows wide-spectrum causality relations. For example, c_{11}="+" of Fig. 74 is an abstract level relation representing only existence of relation between the operator *opr1* and the operand *opd1*; the details of the relation are not given at this level. On the other hand, c_{23}="pop:X" is a concrete level relation in which the operator *opr2* pops a token from the operand *opd3* and unifies the color of the token with X (i.e., X is assigned the color of the token). A designer stepwise refines operators, operands, and causality relations from an abstract level to a concrete level, which is regarded as a design process. The wide-spectrum causality matrix has the following advantages compared with graphical Petri net representations.

- The binary relation in the matrix is more adequate for system element analysis in the earlier design phase. It is much easier for a designer to decide what is a local abstract relation between every two elements than to directly draw a rigid Petri net.
- A spread-sheet editor for the matrix can provide powerful editing abilities. Copying, eliminating, decomposing, browsing, focusing, and checking can be implemented more easily than by a graphical one. Furthermore, graphical representations are difficult to perceive without good topological



operand / operator	opd1	opd2	opd3	opd4		
	state		port		guard	body
		external				
opr1	+		push			
opr2	in		pop:X		X>0	true
opr3		>	+	+		
opr4	+			+		
opr5	+	<	pp			

Figure 74. Causality Matrix

Table 13. Wide-Spectrum Causality Relations

LEVEL	*RELATION*	*EXPLANATION*
Abstract level	$+$	some relation
		blank no relation/undefined
Intermediate level (1)	$<$	input/cause
	$>$	output/effect
Intermediate level (2)	sync	synchronization
	rw	read or write from/to slot
	pp	pop or push from/to port
	trans	state transition
Intermediate level (3)	insync	synchronization with input data
	outsync	synchronization with output data
	read	read from slot
	write	write to slot
	pop	pop from port
	push	push to port
	in	reference of state
	from	transition from state
	to	transition to state
Concrete level	KL1	KL1 code

arrangement. Current automatic topological arrangements in Petri net tools are not yet good enough.

- The wide-spectrum property makes backtracking easier. Backtracking is inevitable in a design process. However, the wide-spectrum property enables the designer to manipulate only common and consistent design documents (causality matrices) throughout the design process, which minimizes the modification efforts in backtracking.

A concrete causality matrix can be transformed straightforward into a process of MENDEL nets. In other words, a causality matrix is a matrix-based representation of MENDEL nets which allows vagueness and informality. Figure 75 shows an example of straightforward transformation from a matrix to a process of MENDEL nets. Remark that each matrix corresponds to one process and does not have a hierarchical structure in itself, which should be represented by MENDEL nets.

operand	free	busy	data	input	output		
	state	state	slot	port	port	guard	body
operator				external	external		
get	from	to	write:X	pop:X		X>0	true
put	to	from	read:X		push:Y	true	Y:=X+1

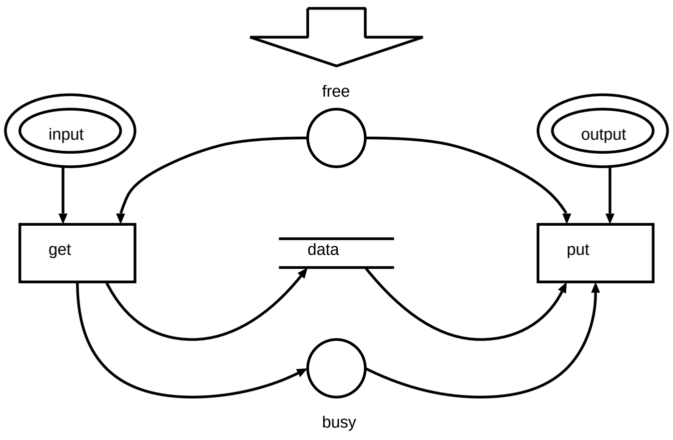


Figure 75. Transformation from a Causality Matrix to a Net

4 Design Method

The proposed design method consists of four phases; design of elementary processes (Phase 1), process interconnection and coordinator creation (Phase 2), design of coordinators (Phase 3), and validation (Phase 4).

First, Phase 1 and Phase 2 are carried out using directly MENDEL net because elementary processes are not so obscure. Then, the design of the coordinator whose structure may be obscure is done by stepwise refinement using the wide-spectrum causality matrix. A procedure of the matrix refinement is summarized as follows.

step 1: method recognition

step 2: port and slot recognition

step 3: state recognition

step 4: functional refinement (writing LPL code)

Finally, MENDEL nets are verified and adjusted using PLTL, and executed visually. Figure 76 outlines the flow chart of the following design methodology.

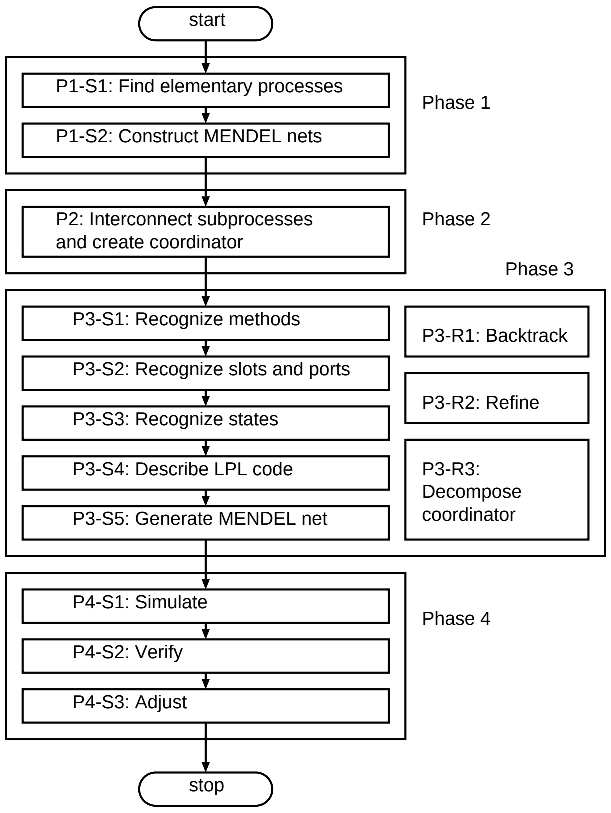

start
P1-S1: Find elementary processes
P1-S2: Construct MENDEL nets
Phase 1
P2: Interconnect subprocesses and create coordinator
Phase 2
Phase 3
P3-S1: Recognize methods
P3-S2: Recognize slots and ports
P3-S3: Recognize states
P3-S4: Describe LPL code
P3-S5: Generate MENDEL net
P3-R1: Backtrack
P3-R2: Refine
P3-R3: Decompose coordinator
P4-S1: Simulate
P4-S2: Verify
P4-S3: Adjust
Phase 4
stop

Figure 76. Design Method (Flow Chart)

[Phase 1]

(Step 1) *Find elementary processes*:

Find all elementary processes whose structure is well defined, and enumerate the methods, slots, and ports for each process. Most hardware-constrained processes are elementary. For example, *cage* is one of the elementary processes in a lift system, and its methods include *move_up, move_down, open_door*, and *close_door.*

(Step2) *Construct a MENDEL net for each elementary process*:

Construct a MENDEL net for each elementary process by appending the state elements and arrows to the methods, slots, and ports listed in step 1. Then, classify each port and method as external one which becomes a plug and is accessed from other processes or internal one.

[Phase 2] *Interconnect elementary processes and create a coordinator*:

Create a new process (parent process) that consists of the elementary processes (subprocesses). Interconnect the plugs of these subprocesses with asynchronous communication and synchronous communication. Some plugs which cannot be directly connected may remain; in this case some process is required to coordinate them. Create a new subprocess (the coordinator) and connect the remaining plugs to it. Note that an initial coordinator has only external ports and methods that are connected to the remaining plugs of the elementary processes.

[Phase 3]

(Step 1) *Create an initial causality matrix and recognize methods*:

Create an initial causality matrix of the coordinator, in which only external ports and methods, listed in phase 1, are filled. Then, recognize all functions (method candidates) of the coordinator and fill them in the matrix.

(Step 2) *Recognize internal ports and slots*:

Judge existence of the causality relation between external/internal methods and external ports, and fill abstract level judgment (+ or blank) in the matrix. Stepwise refine these causality relations into a less abstract level. This stepwise refinement helps a designer to recognize additional internal ports and slots which are necessary to refine the causality relations.

(Step 3) *Recognize state elements*:

Find logical state elements of the coordinator (e.g., *active, sleep, waiting, busy*), and add them and fill causality relations in the matrix. In the causality relation, each method should be decided whether it is enabled or not in each logical state. In addition, consider the partial ordering of method firing, and introduce dummy control state elements to put method firing in order.

(Step 4) *Describe method inscriptions*:

Describe the detailed condition and action for each method by logic programming language. At this point, the causality matrix reaches its most concrete level.

(Step 5) *Generate MENDEL net processes*:

Finally, generate MENDEL net processes of coordinators straightforward from the most concrete-level matrices.

In addition, the following design rules are applicable during Phase 3.

(Rule 1) *Cause design backtracking*:

When any design failures or unexpected functions that require structural rearrangements are detected, go back to any previous steps in Phase 3.

(Rule 2) *Refine methods, slots, and ports*:

Decompose and modify methods, ports, and slots if they have compound functions or meanings.

(Rule 3) *Decompose coordinators*:

Decompose coordinators if the coordinator becomes too large or too complex.

[Phase 4] *Validate constructed MENDEL nets*:

(Step 1) *Simulation*:

Execute MENDEL net by a simulator, and confirm that it works well (i.e., it satisfies your requirements).

(Step 2) *Verification*:

Specify timing constrains for the target system using PLTL which are derived from the original informal specification. Then, verify and analyze whether the constructed MENDEL net satisfies these PLTL constraints (e.g., deadlock-free, interlock).

(Step 3) *Adjustment*:

If the MENDEL net does not satisfy all the specified PLTL constraints, the designer must adjust the MENDEL net to satisfy them, manually using a MENDEL net editor or automatically using the program adjustment mechanism described in Chapter 6.

5 Example: Lift Control System

5.1 Problem

This problem is a revised version of the popular problem presented for the 4th International Workshop on Software Specification and Design [IWSSD 87].

List Control System:
One lift is to be installed in a building with M floors. The problem concerns the logic to move cages between floors according to the following constraints:

- The cage has a set of buttons, one for each floor. These illuminate when pressed and cause the cage to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited by the cage, or when the button is pulled out (canceling the request).
- Each floor has two buttons, one to request an up-lift and another to request a down-lift. These illuminate when pressed. The illumination is canceled when a cage visits the floor and is either moving in the desired direction, or has no outstanding requests. The illumination may also be canceled by pulling the button out (canceling the request).

5.2 Observation of actual design process

The actual design process of the lift system will be traced using MENDELS ZONE. Here, "P_i-S_j:" means Step j of Phase i; it is an index for the design methodology.

(1) **P1-S1:** This lift system has four elementary processes: *cage*, *button*, *floor_button_panel*, and *cage_button_panel*.

(2) **P1-S2:** The constructed MENDEL nets of these elementary processes are shown in Fig. 77. *floor_button_panel*, and *cage_button_panel* are constructed as parent processes of *buttons*. However, since they are very physical and require no coordinators, they ware regarded as elementary processes. Remark, *c* = *cage*, *f* = *floor*, *req* = *request*, *can* = *cancel*, *vis* = *visit*.

(3) **P2:** The top level *lift_system* is constructed by interconnecting 3 subprocesses: *cage*, *floor_button_panel* and *cage_button_panel*. Here are all plugs (*open*, *up*, *down*, *c_req*, *c_can*, *c_vis*, *f_req*, *f_can*, *f_vis*) remain unconnected. Therefore we create a coordinator process, and connect remaining plugs to it (Fig. 78).

(4) **P3-S1:** The initial causality matrix is created. It has 6 external ports and 3 external methods corresponding to the plugs initially connected in (3). We find the following functions (method candidates) of the coordinator and enter them in the matrix:

- *open*: open the door of the cage when the cage visits the requested floor.

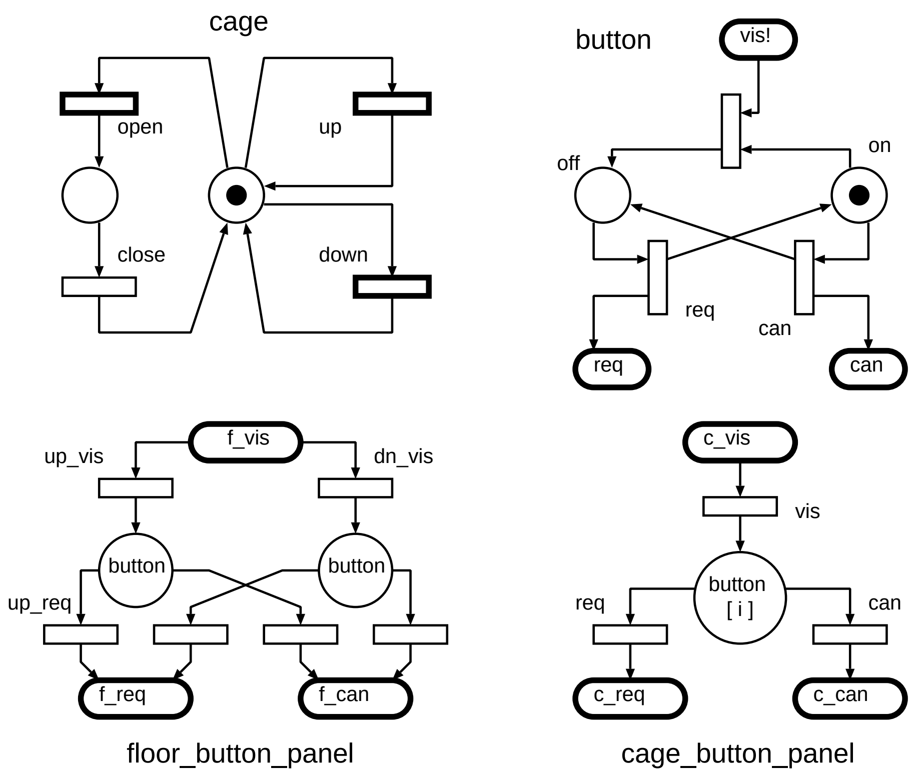


Figure 77. Elementary Processes of Lift Control System

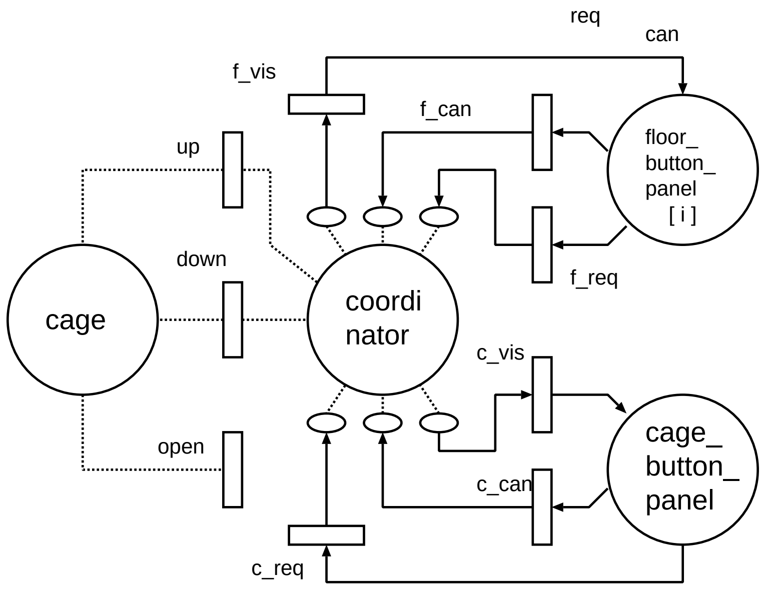


Figure 78. Process Interconnection and Coordinator Creation

- *up*: move up to the target floor.
- *down*: move down to the target floor.
- *c_req* / *f_req*: accept requests from the *cage_button_panel* / *floor_button_panel*.
- *c_can* / *f_can* : delete the canceled floor from the request queue.
- *c_vis* / *f_vis* : acknowledge the *cage_button_panel* / *floor_button_panel* when the cage arrives at the requested floor.

(5) **P3-S2:** Causality relations are entered at the abstract level. We recognize 3 internal slot candidates (*req_que*, *current_f*, *target_f*) and one additional method candidate (start), and add them to the matrix (Fig. 79).

	open	up	down	c_req	c_can	c_bis	f_req	f_can
	method	method	method	port	port	port	port	port
	external	external	external	external	external	external	external	external
open	+							
up		+						
down			+					
c_req				+				
c_can					+			
c_vis						+		
f_req							+	
f_can								+
f_vis								
start								

	f_vis	req_que	current_f	target_f		
	port	slot	slot	slot	guard	body
	external					
open			+	+		
up						
down						
c_req		+				
c_can		+		+		
c_vis			+	+		
f_req		+				
f_can		+		+		
f_vis	+		+	+		
start		+	+			

Figure 79. Causality Matrix (Abstract Level)

- *req_que*: a request queue in which all requests are stored.
- *current_f*: the number of the current floor that the cage is currently staying.
- *target_f*: the number of the target floor to which the cage will go.
- *start*: select a target floor from the request queue.

(6) **P3-R2&S3:** While refining the causality relations stepwise, we refine the operators and operands, and recognize new state elements. The following case shows a fragment of this refinement and state recognition process. When refining the operator *c_can* (Fig. 80(a)), that is a local view of the matrix, we notice that there are two cases.

Case 1: The canceled request remains in *req_que*.

Case 2: The canceled request has already been selected as *target_f*.

Here, we divide *c_can* into *c_can_1* (case 1) and *c_can_2* (case 2), and try to refine each operator. Then, we recognize that the states (active and sleep) are necessary to refine *c_can_2*, and introduce them. Finally, the matrix shown in Fig. 80 (b) is derived.

(a) Original Matrix

	c_can	req_que	target_f		
	port	slot	slot	guard	body
c_can	pop	rw	read		

(b) Refined Matrix

	c_can	req_que	target_f	active	sleep		
	port	slot	slot	state	state	guard	body
c_can_1	pop:X	read:Q1, write:Q2	read:Y	in		X \ = Y	cancel_rfl(X,Q1,Q2)
c_can_2	pop:X		read:Y	from	to	X = Y	true

Figure 80. Causality Matrix (Refinement Example)

(7) **P3-R1:** We notice that we have missed the fact that the cage should stop and open the door at the floor that is stored in the *req_que* even if it is not a current target. Consequently, we must backtrack and modify the design; we must modify the method *open*.

(8) **P3-R3:** The coordinator is divided into 2 coordinators (*cage_controller* and *request_controller*), because the matrix becomes somewhat complex; so it is natural to divide its functions into the cage controller and the request controller. During this dividing, the following interface plugs (external ports) are introduced.

- *start*: acknowledge that the cage has started.
- *end*: acknowledge that the cage has visited the target floor.
- *exit*: acknowledge that the request has been canceled.
- *current*: inform the current floor that the cage is staying.
- *command*: command that the cage opens the door or not.

Divide the matrix into new two matrices which corresponds to generated coordinator processes (*cage_controller* and *request_controller*).

(9) **P3-S3 (for *cage_controller*):** After checking for any method conflicts, we introduce 9 dummy control state elements (*active*, *sleep*, etc.) to serialize the methods (*current*, *command*, *open*, *pass*, *up*, and *down*) to avoid the conflicts.

(10) **P3-S5:** We implement inscriptions used in the methods (e.g., *cancel_rfl* of Fig. 80) which completes the stepwise refinement of the matrix. The final causality matrix of the cage controller is shown in Fig. 81. From this matrix, MENDEL nets of the lift system can be generated automatically. Fig. 82 shows the top level process (*lift_system*), and Fig. 83 shows one of the subprocesses (*cage_controller*). In (b), slots are hide in the display for simplification, and a triangle represents OR-connection of synchronous communication.

(11) We can verify at its skeleton level whether or not this system satisfies the following constraints.

- Deadlock freedom of a cage:
  PLTL formula = $\Box\Diamond(open \lor up \lor down)$
- Once a k-th floor is requested, the cage eventually visits the floor and opens the door unless the request is not canceled:
  PLTL formula = $\Box(((c\_req(k) \lor f\_req(k)) \land \Box(\neg c\_can(k) \land \neg f\_can(k))) \supset \Diamond open(k))$

In this case it is fortunately assured that this system satisfies these constraints in the skeleton level, so no adjustments are necessary.

	open	up	down	start	exit	end	command	current	sleep	active
	method	method	method	port	method	method	port	port	state	state
	external	external	external	external	external	external	external	external		
open	outsync									
end						outsync			to	
exit					insync				to	from
start				pop:X					from	to
check1										from
check2										from
current								push:CF		
pass							pop:COM			
open	outsync						pop:COM			
check3										
check4										
up		outsync								to
down			outsync							to

	target_f	wait_op	end	ch	wait_com	move	up	down
	slot	state	state	state	state	state	port	port
open		from	to					
end			from					
exit								
start	write:X			pop:X				
check1	read:Y	to						
check2	read:Y							
current				from	to			
pass					from	to		
open					from	to		
check3	read:Y					from	to	
check4	read:Y					from		to
up							from	
down								from

	c_floor		
	slot	*guard*	*body*
open		true	true
end		true	true
exit		true	true
start		true	true
check1	read:(X,_)	X=Y	true
check2	read:(X,_)	X\ =Y	true
current	read:CF	true	true
pass		COM=pass	true
open		COM=open	true
check3	read:(X,_)	$X < Y$	true
check4	read:(X,_)	$X > Y$	true
up	read:(X,_),write:(Y,up)	true	Y:=X+1
down	read:(X,_),write:(Y,down)	true	Y:=X-1

Figure 81. Causality Matrix (Concrete Level)

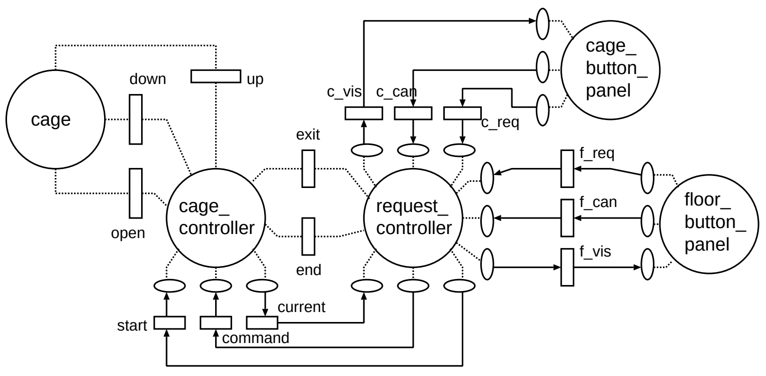


Figure 82. Constructed MENDEL nets (lift_system)

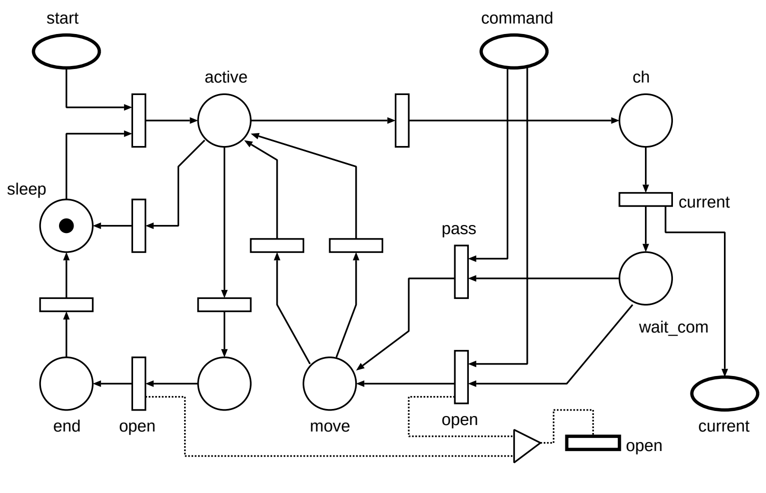


Figure 83. Constructed MENDEL nets (cage_controller)

5.3 Qualitative Evaluation

We briefly show our experiences of the lift system design. When we designed it using only a Petri net editor and with no methodology, we abandoned the use of the editor and did paper works in the earlier design phase because backtracking caused tedious editing and rearrangement. We used a Petri net editor only for a fair copy of the paper works. On the other hand, when we use the causality matrix and the MENDEL net editor according the proposed design method, we succeed to finish design of the lift system without escaping paper works.

6 Related Works

We believe that existing competitors to the proposed design method are RTSAD [Ward 86], OOAD [Booch 94], DARTS [Gomaa 93], PAISLey [Zave 91], STATEMATE [Harel 90], and G-LOTOS [Lee 91]. We have also proposed another design methodology using structured analysis [Honiden 90]. All support data flow, state machine, and hierarchy. The most obvious difference from our approach is that these are not directly based on Petri nets. Each approach has similar abilities in general and distinct merits and demerit in detail. Nevertheless, we favor the Petri-net-based approach from the following reasons.

- Petri net is a multi-paradigm model which can represent both flow model and state model uniformly, while design charts have to be separated into flow model and state model in other design methods. This property is effective to have an accurate grasp of both static and dynamic structure of systems.
- This approach can take advantage of graphical representations and a variety of analysis methods which have been and will be provided by many Petri net researchers.

Recently, several Petri-net-based design methods have been reported. There are two following types in these methods.

- **Compound Type:** Petri net are used only to represent state models instead of other substitutive charts (i.e., state transition diagram or Statechart), where flow models are represented by another chart.
- **Pure Type (Net-Oriented Design Method):** Petri net are used to represent causality relations among system elements, which include both state models and flow models. We call this type of design method *Net-Oriented Design Method.*

Pinci and Shapiro proposed a methodology in which CPN are integrated with SADT (Structured Analysis and Design Technique) [Pinci 91]. SADT is a sophisticated and well-used methodology for requirement analysis [Ross 77]. Bruno also proposed a design method by combination of high-level Petri nets (Protob nets) and structure diagrams (Quid) [Bruno 95] instead of SADT. However, SADT and Quid are based on data flow diagrams and lacks the state transition feature, while our causality matrix supports both data flow and state transition features. Etessami's rule-based design methodology [Etessami 91] is another Petri-net-based approach which uses Abstract Petri Net (APN). A unique feature of APN is the combination of timed and colored Petri net. According to the rule-based design methodology, a designer first formalizes the specification by means of a set of rules and lists attributes representing the status of a target system. Then, he retracts places from the attributes and transitions from the rules, and finally describes APN. Etessami's approach seems to be in the same research direction as our approach. However, Etessami's methodology is less systematic (therefore, has no computer support) and is weaker in design backtracking. In addition, APN is not hierarchical.

The design method by Reisig [Reisig 92] is one of the purest Petri-net-based design method, where it uses two types of Petri nets, Channel Agency net (C/A net) and Individual Token net (I/T net). C/A net is used only to represent static structure of the target system, while I/T net is a simple high-level Petri net and used to static and dynamic structure. Similar to our approach, a C/A net is used in an earlier design phase, then it is refined stepwise into an I/T net. This design method is very interesting and promising. However, it is rough and immature in detail and has plenty of room for improvement. We are now trying to improve this method and combine it with our method [Uchihira 93b].

7 Summary

This chapter has proposed a Petri-net-based design method which utilizes causality matrices and temporal logic. According to this design method, the designer can construct Petri nets by stepwise refinement from an abstract level at the earlier design phase to a concrete level at the final phase. Our method can be classified into net-oriented design methods, where Petri nets are used to represent various causality including data flow, control flow, and state transition. Therefore our method can be distinguished from other popular design methods such as OOAD and RTSAD.

However, net-oriented design methods are not yet mature enough to be used for practical software design as compared with other design methods. We need further experiences and deep consideration about net-oriented design methods.

Chapter 9

MENDELS ZONE: Petri-Net-Based Programming Environment

This chapter describes *MENDELS ZONE*, a Petri-net-based programming environment, which is purposed being suitable for reactive and concurrent systems. MENDELS ZONE adopts MENDEL nets, and provides several utilities including temporal logic verification and adjustment, and computer-aided design method, which are mentioned in the previous chapters.

1 Introduction

Since it is often troublesome for ordinary programmers to produce reactive and concurrent programs as compared with sequential programs, several kinds of CASE (Computer-Aided Software Engineering) tools are inevitable. Requirements for CASE tools for reactive and concurrent systems include editors (graphical and textual), simulators (program simulator, I/O simulator, and environment simulator), debugger, execution monitor, validation tools (verification and analysis), program generater (compiler, translator, and program synthesis), performance evaluation tool, software reuse support tool (program component library), and documentation tool.

MENDELS ZONE [Uchihira 87, Honiden 90, Uchihira 90a] is a programming environment for reactive and concurrent systems, which had been developed over 8 years by 3–7 persons as a part of the Fifth Generation Computer System Project (*FGCS*) [Furukawa 92]. It facilitates the difficult task of concurrent programming for reactive and concurrent systems. MENDELS ZONE adopts a high-level Petri net, MENDEL net, as a kernel programming language. In addition to a MENDEL net editor and a compiler to the concurrent programming language, MENDELS ZONE provides two appealing features described below.

(1) Verification and adjustment using Petri nets and temporal logic

(2) Computer-aided design method for high-level Petri nets

With regard to the latter feature, MENDELS ZONE is unique compared with other Petri net tools because they so far support only the drawing (graphical editor), simulation, and analysis (reachability and invariant analysis) of Petri nets [Feldbrugge90].

The remainder of this chapter is organized as follows. First requirements for CASE tools for reactive and concurrent systems are considered in Section 2. Overview and structure of MENDELS ZONE are given in Section 3. Section 4 illustrates a software development process in MENDELS ZONE. Section 5 introduces a middle-scale example, followed by related works in Section 6.

2 Requirements for Programming Environment

A *programming environment* consists of several constituent tools. The following tools are useful for developing reactive and concurrent systems using Petri nets.

- *Graphical Editor*

  An editor is the most fundamental programming tool. In the Petri-net-based programming environment, a graphical editor is indispensable because the graphical representation of Petri net is one of the most strong points. The recent remarkable progress of graphical user interface technology makes it easier and less expensive to develop sophisticated graphical editors. There still remains some room for consideration in regard to graphical manipulation of hierarchy (module) of Petri nets.

- *Program Synthesizer*

  Program synthesis is defined to generate an executable program from an unexecutable specification such as temporal logic and algebraic specification. It is not realistic to synthesize a whole program in actual software development. Partial program synthesis is a promising solution. Program adjustment is included in this approach.

- *Program Generater (Compiler or Translator)*

  A machine-executable code should be automatically generated from graphical representation of the program. In particular, the *cross-compiler* is required in case that the execution environment differs from the programming environment. Actually, most embedded systems require the cross-compiler.

- *Simulator and Debugger*

  When the execution environment differs from the programming environment, simulation and debugging in the programming environment are very useful. Simulation in the programming environment requires not only a program execution simulator but also an I/O hardware simulator and an environment simulator.

- *Validation Tool*

  Validation tools include a testing tool and a verification tool. *Testing* is easy to apply but difficult to cover all possible cases. On the other hand, *formal verification* can check all possible cases by analyzing the program source code. Verification is very promising for safety-critical reactive systems. However, verification is usually hard to apply ill-structured systems and very expensive. From the practical point of view, complementary use of both testing and verification is effective and necessary [Uchihira 95b].

- *Execution Monitor*

  After testing and debugging the program in the programming environment, it is also necessary to execute and test the program in the actual execution environment as integration testing. In this case, execution monitor is indispensable. Especially visual execution monitor is effective for concurrent systems. We emphasize that testing in the actual execution environment is not so easy because it does not reproducible behavior. For example, errors which appear in the usual execution often disappear when using the monitor (it is called *probe effect*).

- *Performance Evaluation and Real-Time Analysis Tool*

  Performance evaluation is important for some reactive and concurrent systems. Timed Petri net are often used for this purpose. Real-time scheduling analysis (e.g., *rate monotonic analysis*) is also useful for hard real-time systems using real-time operating systems [Stankovic 95].

- *Software Reuse Support Tool*

  In general software reuse is very effective to achieve high productivity. The recent programming environment such as *Visual Basic* and *Visual C++* provide powerful software reuse mechanisms. However, there are some room for consideration in regard to software reuse for reactive and concurrent systems.

- *Requirement Analysis and Design Methodology Support Tool*

  Support tools used analysis and design phases are called *upper CASE tool*, while editor, testing and debugging tools are called *lower CASE tool*. There are many upper CASE tools supporting OOAD and RTSAD. However, most of them are used only for analysis and design phases and are linked to editing, testing and debugging tools. Recently programming environments which integrate both upper CASE tool and lower CASE tool are required and some have been proposed. Petri nets are promising framework to achieve this integrated programming environment.

- *Documentation Tool*

  Some documentation of the final product is necessary for its maintenance. The documentation tool produces documents from program information which can be retracted in the above tools (e.g., graphical editor), or generates documents from the source code by reverse engineering.

3 MENDELS ZONE

MENDELS ZONE is a CASE tool kit for concurrent programming. The kernel concurrent programming language is MENDEL, which is a textual form of a MENDEL net [17] . MENDEL programs are compiled into the concurrent logic programming language *KL1* [Ueda 90, Chikayama 92] and executed on the parallel computer *Multi-PSI* [Taki 89] [18] . MENDEL is regarded as a user-friendly macro language of KL1, whose purpose is similar to *A'UM* [Yoshida 88] and *AYA* [Suzaki 91]. However, MENDEL (MENDEL net) is more convenient for designers to use in designing state-transition-based reactive and concurrent systems.

MENDELS ZONE provides the following facilities.

- Automatic generation of MENDEL elementary processes from algebraic specification [Honiden 91a],
- MENDEL-net-based programming environment backed up by design methodology [Uchihira 92b], and
- Verification and adjustment tool using bounded MENDEL nets and temporal logic [Uchihira 95a].

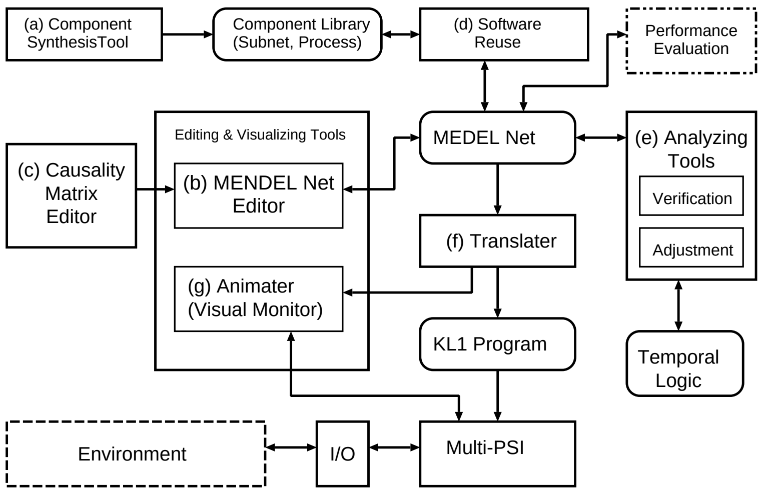


Figure 84. MENDELS ZONE (Block Diagram)

Figure 84 shows a block diagram of MENDELS ZONE. These tools are implemented on Multi-PSI except for a causality matrix editor. We have also proposed a performance evaluation tool [Honiden 94] for MENDEL programs. However, it was not yet implemented in MENDELS ZONE. Table 14 shows how MENDELS ZONE satisfies requirements of CASE tools enumerated in the previous section.

Figure 85 shows the graphical user interface of MENDELS ZONE, which consists of several subwindows. The designer basically constructs MENDEL nets and execute them through this interface. The following sections describe constituent tools of MENDELS ZONE in detail.

[17] MENDELS ZONE supports an only subset of MENDEL net, because Multi-PSI is a symbol manipulation machine, not a real-time control machine. For example, MENDELS ZONE omits scheduling facilities of MENDEL net.

[18] MENDEL programs can also be translated into the C language and be executed on a distributed personal computer system [Uchihira 89b].

Table 14. Requirements for CASE tools and MENDELS ZONE

Requirements	*MENDELS ZONE*
Graphical Editor	(b)
Program Synthesizer	(a),(e)
Program Generator	(f)
Simulator/Debugger	—
Validation Tool	(e)
Execution Monitor	(g)
Performance Evaluation	—
Reuse Support	(d)
Design Methodology	(c)
Documentation	—

(a) — (g) indicate functions of MENDELS ZONE which are shown in the block diagram (Fig. 84).

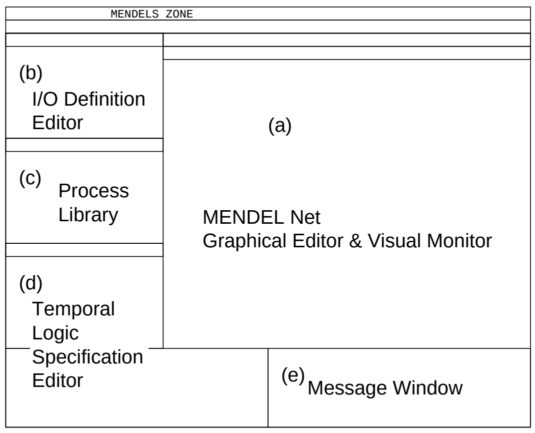

MENDELS ZONE
(b)
I/O Definition Editor
(c)
Process Library
(d)
Temporal Logic Specification Editor
(a)
MENDEL Net Graphical Editor & Visual Monitor
(e)
Message Window

Figure 85. MENDELS ZONE (Graphical User Interface)

3.1 Graphical MENDEL Net Editor

- *MENDEL Net Editor*

  The designer constructs each process of MENDEL nets using a graphic editor (Fig. 85(a)) which provides the creation, deletion, and placement functions for ports, state elements, flags, slots, methods, arrows, and tokens. This editor also supports the hierarchical expansion and reduction of nets, and the transition over the process-oriented hierarchy (i.e., from process to subprocess, and vice versa).

- *Method Editor*

  The method editor provides several editing functions specific to a high-level Petri net. Using the method editor, the designer describes methods (their conditions and actions) in detail with KL1. Furthermore, the editor checks syntax and consistency of edited methods. This method editor is activated by clicking the target method in the MENDEL net editor.

- *I/O Definition Editor*

  This I/O definition editor (Fig. 85(b)) is used to assign I/O devices to I/O places. MENDELS ZONE provides the following I/O devices.

  - *Files*: The program reads/writes character streams from/to designated files.
  - *Windows*: The program reads/writes character streams from/to standard input/output windows.
  - *Lists*: The program reads character streams represented as lists (e.g., $[a, b, c, d, ...]$).

3.2 Causality Matrix Editor

This spread-sheet editor supports stepwise refinement of the *causality matrix*. It provides the following functions:

- creation, deletion, and renaming of methods, ports, slots, and state elements,
- dividing methods, ports, slots, and state elements into detailed ones,
- checking whether refined relations are legal, and
- localizing and focusing the view of relations of designer's interest.

The causality matrix editor is implemented on a UNIX workstation and consists of 3 parts; a general purpose spread-sheet editor (*Oleo*), a consistency checker, and a translator.

- Oleo: a free software for spread-sheet editing.
- Checker checks whether the matrix data edited by Oleo are consistent.
- Translator translates the matrix data into MENDEL nets.

3.3 Software Reuse Support Tool

Reusable processes are stored in the *process library* (Fig. 85(c)). This library tool supports browsing and searching. In MENDELS ZONE, processes can be not only retrieved and but also interconnected in two main ways; manually and automatically.

Manual Interconnection

The designer selects an process from an process library and interconnects these processes with arrows manually. These operations are carried out graphically using the MENDEL net editor.

Automatic Interconnection

The designer gives program specifications as a set of *input/output attributes* which are a kind of I/O data type. Appropriate processes are then selected from an process library and interconnected automatically by pattern matching of these I/O attributes. Automatic retrieval and interconnection are carried out, according to the following principles [Uchihira 87, Honiden 94].

(1) A pair of plugs having the same attributes can be interconnected.

(2) All required output attributes must be reachable from given input attributes through connected processes and arrows.

These automatic retrieval and interconnection can be formalized as a simple and classical *planning problem* in Artificial Intelligence. For example, when the following attributes are given, process **B**, **C**, and **D** are retrieved and interconnected as shown in Fig. 86.

- *Input attribute* **a**, **b** ;
- *Output attribute* **e** ;

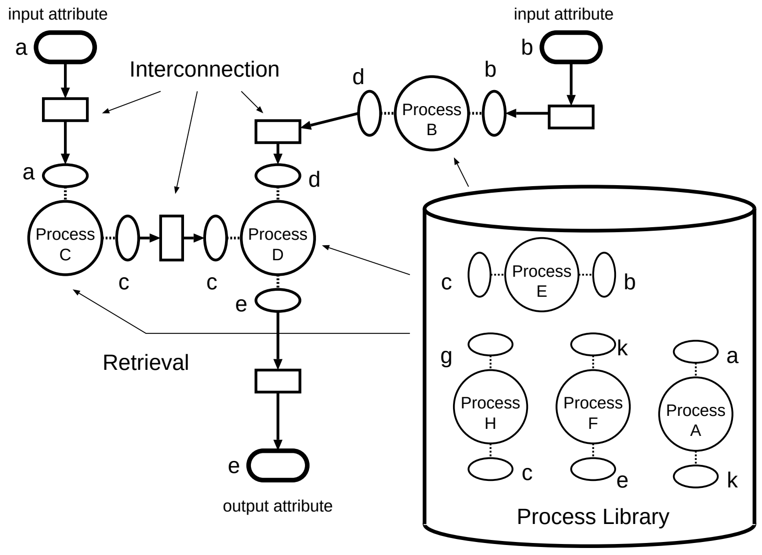


Figure 86. Automatic Process Retrieval and Interconnection

More Flexible Automatic Interconnection

This automatic retrieval and interconnection (i.e., automatic binding) seems to be not powerful enough. The binding mechanism depends on the simple pattern matching between output and input attribute names. In some cases, it might find no candidate to fit the given I/O attributes, or a lot of candidates in other cases. More information must be needed to select the most adequate candidate. To overcome this problem, we adopt a kind of semantic network (called *attribute network*) which represents the attribute structure and define a metric to order the candidates on the semantic network. Detail techniques are described in [Uchihira 87].

3.4 Verification and Adjustment Tool

In the previous chapters, compositional verification and adjustment was investigated for transition systems. When MENDEL nets are supposed to be bounded, verification and adjustment can be also applied to MENDEL nets, because bounded MENDEL nets are equivalent to transition systems. Therefore, only skeletons of MENDEL net structures are automatically retracted (detailed KL1 codes of methods are ignored) in MENDELS ZONE. Furthermore, every asynchronous communication should be approximated by bounded buffers.

Then the verification tool checks whether a MENDEL net satisfies the given PLTL constraints entered by the designer using the *PLTL editor* (Fig. 85(d)). If the net fails to satisfy the constraints, the adjustment tool can automatically adjust (tune up) the net to satisfy the PLTL constraints by adding an arbiter process [Uchihira 90a, Uchihira 95a].

We note that PQL is not used and PLTL is used as a specification language in MENDELS ZONE. The reason is that the same specification language must be used for both verification and adjustment. Unfortunately, we do not provide compositional adjustment method for PQL. To be exact, we do not try to provide it because we think branching time temporal logic including PQL is ill-suited for synthesis and adjustment. Verification method for bounded Petri nets and temporal logic is a special case of our method proposed in Chapter 4. Moreover, we can also use other efficient verification methods for PLTL after bounded Petri nets are translated to transition systems. For example, a model checking method for PLTL [Vardi 86] is one of the most efficient one.

The verification and adjustment are based on the theorem proving method (i.e., tableau construction) of PLTL that is efficiently executed on Multi-PSI. The basic idea of parallel graph generation algorithm which is used in generating finite state processes from PLTL is shown in [Patent KOUKAI H4-259071].

3.5 Program Execution on Multi-PSI

The adjusted MENDEL net is translated into its textual form (MENDEL program). The MENDEL program is compiled into a KL1 program by the *MENDEL translator.* The generated KL1 program can be executed on Multi-PSI. Each Process of the MENDEL program may run on the different CPU. Several compilation techniques (e.g., separate compilation) are introduced here to deal with large-scale programs. During execution, firing methods blink on the *visual monitor* (Fig. 85(a)), and the values (colors) of the tokens are displayed on the message window (Fig. 85(d)). The designer can visually check that the program behaves satisfactorily.

4 Software Development Process in MENDELS ZONE

Fig. 87 (Data Flow Diagram) and Fig. 88 (Flow Chart) show a typical software development process in MENDELS ZONE. The designer should construct a target program according to the following steps.

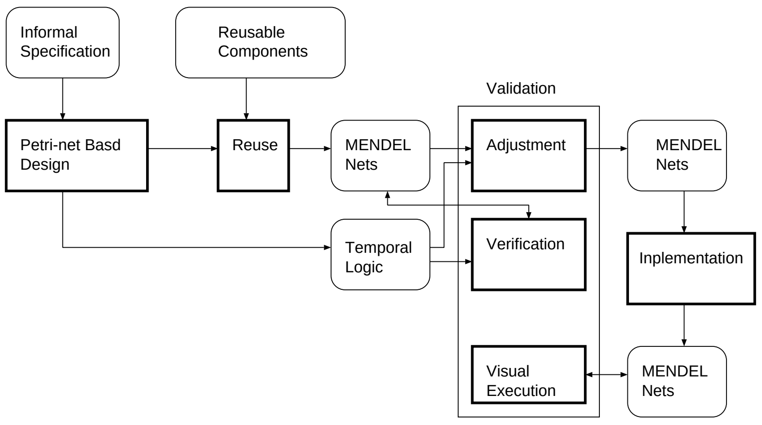


Figure 87. Software Development Process in MENDELS ZONE (Data Flow Diagram)

(Step 1): *MENDEL Net Construction*

A designer constructs a MENDEL net using the MENDEL net editor and the process library as follows.

- **(Step 1-1)** Construct elementary MENDEL processes basically by software reuse, where MENDELS ZONE provides a process library and a process retrieval tool. If the library has no suitable reusable MENDEL processes, MENDELS ZONE can synthesize it from a given algebraic specification. It is also possible for the designer to construct the elementary MENDEL process by himself using the MENDEL net editor.

- **(Step 1-2)** Interconnect MENDEL processes by communication links using the graphic editor to make a new compound MENDEL process. A large-scale program can be constructed in this compositional way. The designer can also make use of an automatic process interconnection mechanism provided in MENDELS ZONE.

Constructed programs are functionally-correct temporally-imperfect (FTCI) because a designer reuses programs whose possible behaviors he may not fully understand; so communication links may be incomplete.

(Step 2): *MENDEL Net Verification and Adjustment*

The compositional adjustment is used in cooperation with the verification. In MENDELS ZONE, the designer first finds existing bugs by the verification step, then adjusts the program to remove the bugs by the adjustment step.

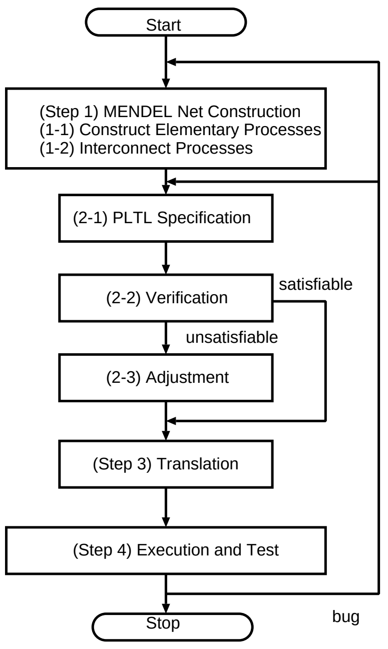

Start
(Step 1) MENDEL Net Construction
(1-1) Construct Elementary Processes
(1-2) Interconnect Processes
(2-1) PLTL Specification
(2-2) Verification
satisfiable
unsatisfiable
(2-3) Adjustment
(Step 3) Translation
(Step 4) Execution and Test
Stop
bug

Figure 88. Software Development Process in MENDELS ZONE (Flow Chart)

After constructing an FCTI MENDEL net, the designer specifies safety and liveness properties that must be satisfied by MENDEL net. These properties are specified by temporal logic.

The verification and adjustment procedure in MENDELS ZONE is as follows.

- **(Step 2-1)** The designer gives a PLTL formula for a MENDEL net of each elementary or compound process.
- **(Step 2-2)** MENDELS ZONE checks whether a MENDEL net satisfies a given PLTL formula.
- **(Step 2-3)** When it does not satisfy the PLTL formula, the adjustment method is invoked.

(Step 3) and (Step 4): *Concurrent Program Generation and Execution*

The adjusted MENDEL program is compiled into a KL1 program, which can be executed on Multi-PSI. The designer can check visually that the adjusted program satisfies his expectation. If not, he should consider two types of bugs.

- Bugs in the temporal logic constraints, and

- Bugs in the KL1 code attached to transitions (i.e., its enable conditions and additional actions), which are ignored in translating to FSP.
- Bugs hidden when translating unbounded MENDEL nets to bounded nets.

5 Example: Power Plant Control System

Using MENDELS ZONE, we have constructed and evaluated several small-scale and middle-scale reactive and concurrent systems including a lift control system, a machine control system for processing (i.e., etching) printed circuit boards [19] , and a control system for a power plant [20] . This section explains a middle-scale example of the power plant (Fig. 89) in detail.

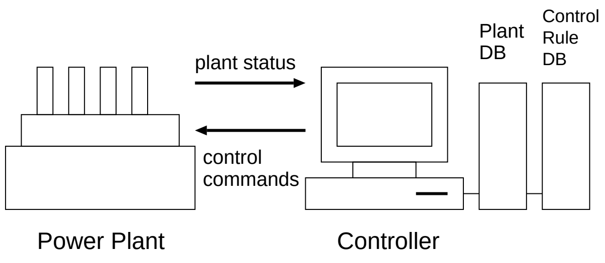


Figure 89. Power Plant Control System

A requirement of the system is summarized as follows. A controller observes plant status continuously and periodically, and in response to changes of plant status it selects control commands according to the control rules, then the controller sends commands back to the plant. This control cycle consists of the following steps.

1. The controller periodically watches current plant status and updates a plant database.
2. Changes of plant status is detected by comparing the current status with the previous status in the plant database.
3. The most appropriate control rule is selected based on changes of status from a control rule database.
4. The selected rule is applied to derive control goals.
5. To achieve the control goal, concrete commands are computed and sent to the plant.

First, the designer decomposes the control software into 6 elementary processes (Fig. 90).

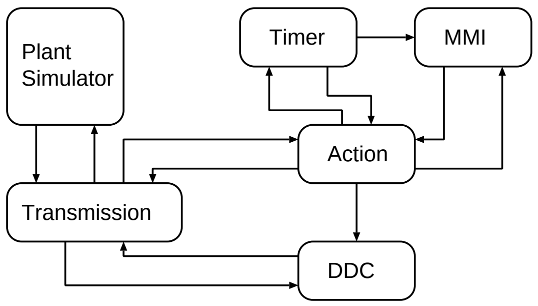


Figure 90. Power Plant Control System (Process Structure)

[19] This was demonstrated at The National Fifth Generation Computer System Symposium 1991.
[20] This was demonstrated at International Conference on Fifth Generation Computer Systems 1992 [FGCS 92].

- Plant process (Controlled Objects)
  This process provides current plant status to the controller, receives control commands from the controller and changes the status according to the commands. In our experimental system on Multi-PSI, a plant simulator is used instead of an actual plant.
- Transmission process
  This process transmits plant status and control commands from/to an action process and a DDC process to/from the plant process.
- Action process
  The current plant status and previous plant status stored in the plant database are compared in this process. The action process detects changes of the plant status, then decides which control rule is applied. By evaluating the rule, a control goal is derived.
- DDC (Direct Digital Control) process
  To achieve the control goal given by the action process, concrete commands are computed and sent to the transmission process using DDC.
- Timer process
  This process provides real-time management to other processes.
- MMI (Man-Machine Interface) process
  This process provides the operator interface which includes plant status monitoring and the operator instruction handling.

First, each process is constructed by the MENDEL net editor. Here, only the action process is explained in detail. We suppose that the designer initially construct a MENDEL net of the action process shown in Fig. 91 where data flow among ports, slots, and methods is specified but control flow is missing. Control flow is latter synthesized by program adjustment from temporal logic specification.

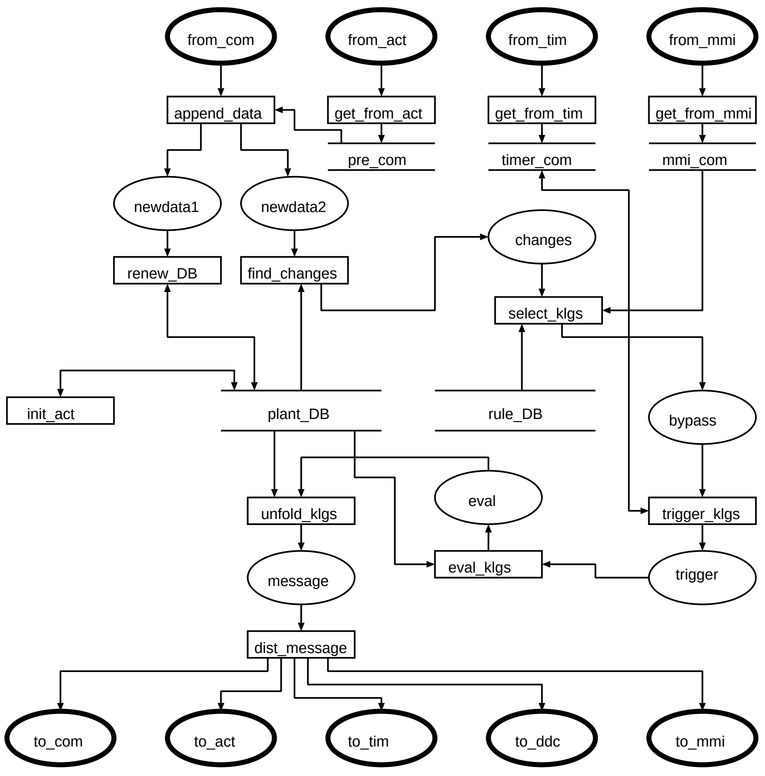


Figure 91. MENDEL net of action process

The action process includes several methods to access the plant database. Some of methods are shown in the textual form as follows. An *action part* of each method is described by KL1 predicates (e.g., `generate_new_data`), which are defined in a junk part.

```
method(append_data,_,[from_com(Data),pre_com(Com)],[newdata1(ND1),newdata2(ND2)]) :-
     true | generate_new_data(Data,Com,ND1,ND2).
method(find_changes,_,[newdata2(ND),plant_DB(PDB)],[changes(CL)]) :-
     true | detect_changes(ND,PDB,CL).
method(renew_DB,_,[newdata1(ND1),plant_DB(PDB1)],[plant_DB(PDB2)]) :-
     true | database_update(ND1,PDB1,PDB2).
method(select_klgs,_,[changes(CL),rule_DB(RDB),mmi_com(MC)],[bypass(KLGS)]) :-
     true | select_klgs_from_rdb(MC,CL,RDB,KLGS).
method(trigger_klgs,_,[timer_com(TC1),bypass(KLGS)],[timer_com(TC2),trigger(KLGS)]) :-
     TC1 = ok | TC2 = nil.
method(eval_klgs,_,[trigger(KLGS),plant_DB(PDB)],[eval(NewGoal)]) :-
     true | evaluate_klgs(KLGS,PDB,NewGoal).
```

For this process, the designer can specify temporal logic specification using the PLTL editor. The specification is mainly related to access control of the plant database so as to preserve consistency of data.

- $\neg$ `append_data` U `init_act`
  Before initializing the plant database, no access to the database is permitted.
- $\Box($ `append_data` $\supset \bigcirc(\neg$ `renew_DB` U `find_changes` $))$
  The plant database must not be renewed before checking the changes between current and previous status.
- $\Box($ `append_data` $\supset \bigcirc(\neg$ `trigger_klgs` U `renew_DB` $))$
  After updating the plant database, rule evaluation is triggered off.
- $\Box($ `append_data` $\supset \bigcirc(\neg$ `append_data` U `eval_klgs` $))$
  Getting a new plant status is not permitted before rule evaluation is finished.

In this case, these constrains are obviously not satisfied, then MENDELS ZONE adjusts the process automatically. The adjustment does not require large computing cost (within 1 minute). As a result, 5 state elements are added to the original MENDEL net to satisfy temporal logic specification. The adjusted process can be watched in Fig. 92. In this figure, added state elements are painted halftone, and slots are not displayed for simplicity.

After all elementary processes are constructed, they are interconnected with the MENDEL net editor. Figure 93 shows a top-level process in MENDELS ZONE which corresponds to the process structure of Fig. 90. In the top level, the designer can verify and adjust a MENDEL net again. In this example, he verifies whether it is deadlock free. Actually, several deadlock states are detected. Since they are due to bugs of elementary processes, he does not use the program adjustment at top level and debugs them manually. A size of a final MENDEL net amounts to 4300 lines in MENDEL textual representation.

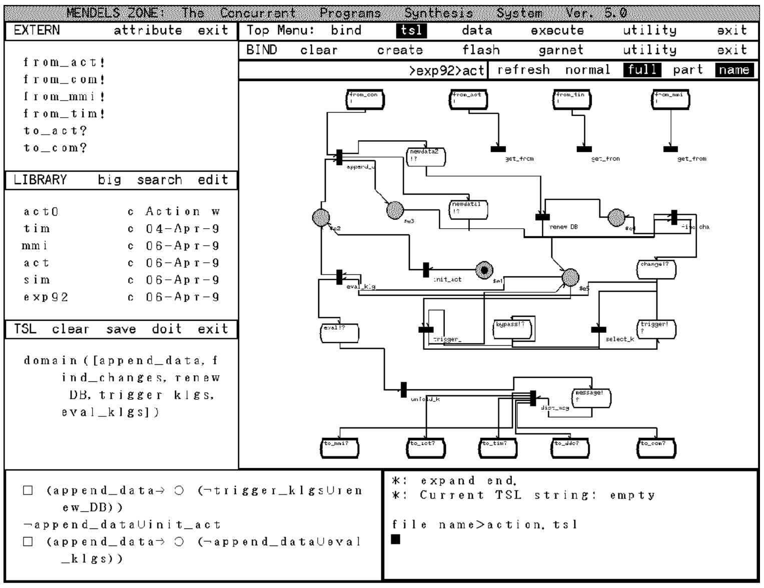
Finally, all processes (MENDEL program) are compiled into KL1 codes (6200 lines) and executed and monitored visually on MENDELS ZONE. It is reported that development cost is cut down to half compared with the case that the designer implemented the same system using naked KL1 and Multi-PSI [Uraoka 92]. This cost-down results from reduction of debugging efforts because debugging of the naked KL1 program is troublesome for ordinary programmers.

6 Related Works

6.1 Comparison with STATEMATE

STATEMATE is another CASE tool for reactive and concurrent systems, in which three types of charts (*module-chart*, *activity-chart*, and *Statechart*) are written by the designer. These charts correspond to elements of MENDEL net as shown in Table 15.

We compare MENDELS ZONE with STATEMATE in regard to the following items.



```
from_act!
from_com!
from_mmi!
from_tim!
to_act?
to_com?
```

```
act0        c Action w
tim         c 04-Apr-9
mmi         c 06-Apr-9
act         c 06-Apr-9
sim         c 06-Apr-9
exp92       c 06-Apr-9
```

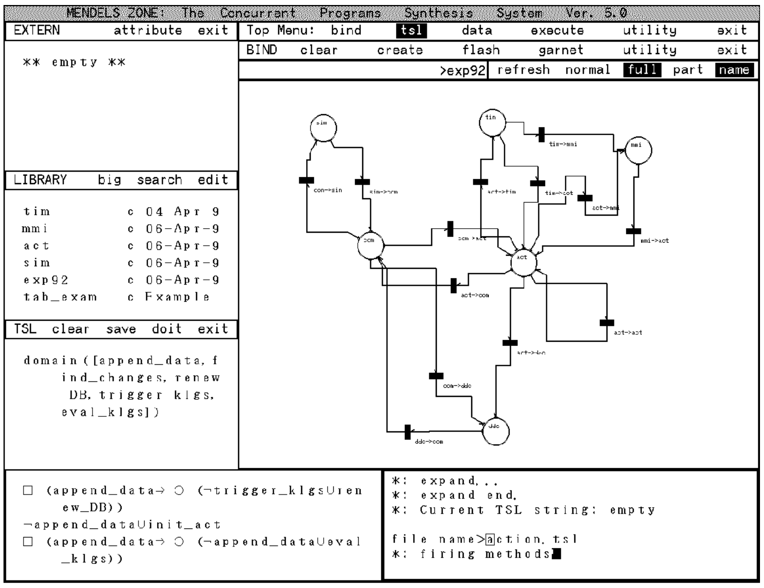
```
domain([append_data, f
     ind_changes, renew
      DB, trigger klgs,
     eval_klgs])
```

```
□ (append_data→ ○ (¬trigger_klgs∪ren
    ew_DB))
¬append_data∪init_act
□ (append_data→ ○ (¬append_data∪eval
    _klgs))
```

```
*: expand end.
*: Current TSL string: empty

file name>action.tsl
■
```

Figure 92. MENDELS ZONE (Adjusted Action Process)



```
** empty **
```

```
tim         c 04 Apr 9
mmi         c 06-Apr-9
act         c 06-Apr-9
sim         c 06-Apr-9
exp92       c 06-Apr-9
tab_exam    c Example
```

```
domain([append_data, f
     ind_changes, renew
      DB, trigger klgs,
     eval_klgs])
```

```
□ (append_data→ ○ (¬trigger_klgs∪ren
    ew_DB))
¬append_data∪init_act
□ (append_data→ ○ (¬append_data∪eval
    _klgs))
```

```
*: expand...
*: expand end.
*: Current TSL string: empty

file name>action.tsl
*: firing methods█
```

Figure 93. MENDELS ZONE (Top-level MENDEL net)

Table 15. STATEMATE vs. MENDELS ZONE

STATEMATE	*MENDELS ZONE*
module (module-chart)	process
control activity (activity-chart)	coordinator
activity (activity-chart)	method
data store (activity-chart)	slot, port
state (Statechart)	state
data flow	arrow
control flow	arrow
inscription language	KL1

- **State Transition Model Complemented by Data Flow**

  Both MENDELS ZONE and STATEMATE adopt an extended state transition model (i.e, Petri net and Statechart) which plays a key part in reactive control systems, while a data flow model is also utilized complementarily. This approach is favorable in most CASE tools for reactive and concurrent systems.

- **Viewpoint**

  STATEMATE adopts three type of charts (module-chart, activity-chart, Statechart) and supports a *multi-view design method* using them, while MENDELS ZONE adopts one type of chart (MENDEL net) and supports a *single-view design method.* The multi-view design is suited for analyzing ambiguous requirements in the earlier design phase and understanding outline of the systems in the maintenance phase. However, it is difficult to keep consistency among three charts. On the other hand, the single-view design is suited to trace dynamic behaviors in the testing and debugging phase because design information is represented uniformly and consistently. However, it is not easy to describe MENDEL nets in the earlier design phase; a causality matrix is introduced in MENDELS ZONE.

- **Hierarchy**

  While MENDEL net adopts only process-oriented hierarchy, STATEMATE provides several types of hierarchy; concept-oriented hierarchy in activity-chart which is useful for top-down stepwise refinement, and state-oriented hierarchy in Statechart which is useful to specify exception handling. On the other hand, STATEMATE does not much care about process structure as compared with MENDELS ZONE.

- **Analysis**

  STATEMATE provides a simulation and a global state analysis tool, while MENDELS ZONE provides verification and adjustment using temporal logic. Verification ability of temporal logic is stronger than global state analysis in STATEMATE. However, the analysis tools of STATEMATE is more sophisticated from user's point of view.

- **Code Generation**

  STATEMATE generates C programs or Ada programs from the described three charts, while MENDELS ZONE generates KL1 programs from MENDEL nets.

From the above consideration, we can summarize comparison between MENDELS ZONE and STATEMATE as follows. MENDELS ZONE and STATEMATE are generally similar, but their target in software development process is different. MENDELS ZONE puts emphasis on the design phase where integration and tractability with a single chart are important, while STATEMATE puts emphasis on the analysis phase where it is important to collect functions and arrange them in order from three types of charts.

6.2 Comparison with Other Petri Net Tools

Most Petri net tools (DESIGN/CPN, GreatSPN, Cabernet, etc.) are used only for modeling, simulating, and analyzing systems in a prototyping phase. After finishing the prototyping phase, a target concurrent program for an actual reactive and concurrent system has to be manually reconstructed in an implementation phase. Some tools can generate program source code written by standard languages. However, these are sequential programming languages. For example, Design/CPN provides the automatic generation of SML (Standard ML) codes [Jensen 92], where SML is not a concurrent programming language. MENDELS ZONE can automatically generate target concurrent programs directly from high-level Petri nets where generated programs run efficiently in the real parallel and distributed environment.

7 Summary

We have presented an overview and a system structure of MENDELS ZONE and how to construct concurrent programs using MENDELS ZONE. MENDELS ZONE is just an experimental system in order to adopt and evaluate novel technologies developed by us (verification, adjustment, design method). Although it is immature, it is reported that in the middle-scale example the designer can construct concurrent programs easier in MENDELS ZONE than he does it with naked KL1.

Since Multi-PSI is designed for parallel symbol manipulation and not for reactive systems, MENDELS ZONE is week in implementation accommodated to the actual reactive and real-time environment. We shall implement another version of MENDELS ZONE on the other platform in future.

Chapter 10

Conclusion

1 Review of Developments

I have proposed software development techniques for reactive and concurrent systems using Petri net and temporal logic. These techniques include specification, verification, synthesis, and design methodology. Particularly, to put verification and synthesis into practical, I introduced the compositional program verification and compositional program adjustment. To embody and evaluate these techniques, I have also developed a programming environment, MENDELS ZONE. MENDELS ZONE is available to the public as ICOT Free Software, accessible via Internet (http://www.icot.or.jp/ICOT/IFS/ifs.html)

2 Current Status

MENDELS ZONE shows a typical process and a CASE environment for software development using Petri net and temporal logic. I believe that MENDELS ZONE can play an important role as a reference prototype of CASE tools for reactive and concurrent systems. However, since MENDELS ZONE is no more than the prototype, practical domain-specific CASE tools should be reconstructed based on MENDELS ZONE for actual software development. In fact, we have been constructing a CASE tool for chemical plant control systems (SAVE/SFC, ref. Chapter 5, Section 5.3), in which several techniques of MENDELS ZONE have been adopted.

Of course, some of techniques proposed in this thesis are immature and inexperienced for the actual software development. The relation of Petri net and temporal logic is well researched and mature from the theoretical point of view. However, from the practical point of view, there is still room for improvement in the efficient verification and adjustment techniques. Moreover, Petri-net-oriented design methodology is a unexplored research subject, and our work is nothing but one of trail-blazing efforts. *Three phase net-oriented software design method* [Uchihira 97a] which we recently proposed is another trail-blazing effort.

3 Future Works

Our future works can be summarized into the following directions.

- **Sophisticated Specification Language**
  - **High-level Petri Net:** Although a proposed High-level Petri net (MENDEL net) has a sufficient expressive power, there is still room for improvement in easiness of describing the actual systems. An introduction of domain-specific macro-expressions seems to be a shorter way to the solution.
  - **Temporal Logic:** Although several extended temporal logics handling real-time have been proposed, there is no extended temporal logics handling controllability and observability explicitly, which are essential feature of reactive systems. We are doing research on the temporal logic extended to handle controllability and observability.

- **Composite Program Verification**

  To be concerned with efficient program verification based on model checking, there are other approaches besides compositional verification. One of them is a *partial order approach*, which is full of promise and well investigated these years. It is practical to use compositional method and partial order method case by case and complementarily mentioned in Chapter 5. Furthermore, it is promising to harmonize and integrate both verification and conventional validation techniques like testing/simulation in the common CASE environment. *Hypersequential programming* [Uchihira 96c, Uchihira 97b, Uchihira 97c] is our latest challenging proposal in this direction. Hypersequential programming is intended to make actual concurrent programs highly reliable by conventional testing which is strengthened by verification techniques.

- **Cultivation of Adjustment and Petri-Net-Based Design Methodology**

  Since program adjustment and Petri-net-based design methodology which I proposed belong to pioneer's work, it is very important to cultivate the subjects from now on. In particular, it is necessary to apply these methods to practical examples, evaluate, and improve them.

Bibliography

[Abadi 89] M. Abadi, L. Lamport, P. Wolper, Realizable and Unrealizable Specifications of Reactive Systems, *16th International Colloquium on Automata, Languages, and Programming (ICALP)*, *Lecture Notes in Computer Science*, Vol.372, Springer-Verlag, 1989.

[Andrews 83] G.R. Andrews and F.B. Schneider, Concept and Notations for Concurrent Programming, *ACM Computing Surveys*, Vol.15, No.1, 1983.

[Alur 89] R. Alur and T.A. Henzinger, A Really Temporal Logic, *Proc. IEEE 30th Annual Symp. on Foundations of Computer Science (FOCS)*, 1989.

[Alur 90] R. Alur, C. Courcoubetis, D. Dill, Model-Checking for Real-Time Systems, *Proc. 5th IEEE Symp. on Logic in Computer Science (LICS)*, 1990.

[Alur 91] R. Alur and D. Dill, The Theory of Timed Automata, *Real-Time: Theory and Practice*, *Lecture Notes in Computer Science*, Vol.600, Springer-Verlag, 1991.

[Arnold 92] A. Arnold, *Finite Transition Systems*, Masson (in French), 1992, Prentice Hall (in English), 1994.

[Andersen 92] H.R. Andersen and G. Winskel, Compositional Checking of Satisfaction, *Conference on Computer-Aided Verification (CAV'91)*, *Lecture Notes in Computer Science*, Vol.575, Springer-Verlag, 1991.

[Battiston 88] E. Battiston, F. de Cindio, G. Mauri, OBJSA Nets: A Class of High-Level Nets having Objects as Domains, *Advances in Petri Nets*, *Lecture Notes in Computer Science*, Vol.340, Springer-Verlag, 1988.

[Barringer 84] H. Barringer, R. Kuiper, A. Pnueli, Now You May Compose Temporal Logic Specifications, *Proc. 16th ACM Symp. on Theory of Computing (STOC)*, 51–63, 1984.

[Beaten 90] J. Beaten, ed., *Application of Process Algebra*, Cambridge Univ. Press, 1990.

[Bellettini 93] C. Bellettini, M. Felder, and M. Pezze', Merlot: A Tool for Analysis for Real-Time Specifications, *Proc. of 7th Internat. Workshop on Software Specification and Design (IWSSD)*, 1993.

[Ben-Ari 83] M. Ben-Ari, A. Pnueli, Z. Manna, The Temporal Logic of Branching Time, *Acta Informatica*, **20**, 1983.

[Bernstein 81] F.A. Bernstein, N.Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys*, Vol.13, No.2, 1981.

[Berthomieu 91] B. Berthomieu and M. Diaz, Modeling and Verification of Time Dependent Systems Using Time Petri Nets, *IEEE Trans. on Software Engineering*, Vol.17, No.3, 1991.

[Boehm 76] B.W. Boehm, Software Engineering, *IEEE Trans. on Computers*, Dec., 1976.

[Bolognesi 89] , T. Bolognesi and D.Latella, Techniques for the Formal Definition of the G-LOTOS Syntax, *1989 IEEE Workshop on Visual Languages*, 1989.

[Booch 94] G. Booch, *Object Oriented Design with Application*, (second edition), The Benjamin/Cummings Publishing, 1994.

[Boudol 89] G. Boudol, V. Roy, R. de Simone, D. Vergamini, Process Calculi, From Theory to Practice: Verification Tools, *INRIA Report*, No.1098, 1989.

[Bradfield 92] J. C. Bradfield, *Verifying Temporal Properties of Systems*, Birkhäuser , 1992.

[Brauer 91] W.Brauer, R.Gold, W.Vogler, A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets, *Advances in Petri Nets 1990*, *Lecture Notes in Computer Science*, Vol.493, Springer-Verlag, 1991.

[Bruno 95] G. Bruno, *Model-based Software Engineering*, Chapman & Hall, 1995.

[Büchi 62] J.R. Büchi, A decision method in restricted second order arithmetic, *Proc. Internat. Congr. Logic, Method. and Philos. Sci.*, 1960, also Stanford University Press, 1962.

[Burch 90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, J. Hwang, Symbolic Model Checking: 10^{20} States and Beyond, *Proc. 5th IEEE Symp. on Logic in Computer Science (LICS)*, 1990.

[Cherkasova 87] L.A.Cherkasova and V.E.Kotov, The Undecidability of Propositional Temporal Logic for Petri Nets, *Computers and Artificial Intelligence* Vol.6, No. 2, 1987.

[Chikayama 88] T. Chikayama, et al. , Overview of the Parallel Inference Machine Operating System (PIMOS), *Proc. of Internat. Conf. on Fifth Generation Computer Systems 1988 (FGCS88)*, ICOT, 1988.

[Chikayama 92] T. Chikayama, Operating System PIMOS and Kernel Language KL1, *Proc. of Internat. Conf. on Fifth Generation Computer Systems 1992 (FGCS92)*, ICOT, 1992.

[Christensen 92] S.Christensen, L.Petrucci, Towards a Modular Analysis of Coloured Petri Nets, *Proc. 13th Internat. Conf. on Application and Theory of Petri Nets (ICATPN)*, *Lecture Notes in Computer Science*, Vol.616, Springer-Verlag, 1992.

[Christensen 94] S.Christensen, N.D. Hansen, Coloured Petri Nets Extended with Channels for Synchronous Communication, *Proc. 15th Internat. Conf. on Application and Theory of Petri Nets (ICATPN)*, *Lecture Notes in Computer Science*, Vol.815, Springer-Verlag, 1994.

[Clarke 82] E.M. Clarke, and E.A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic, *Lecture Notes in Computer Science*, Vol.131, Springer-Verlag, 1982.

[Clarke 86] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 2, 1986.

[Clarke 87] E.M. Clarke and O. Grümberg, Reaserch on Automatic Verification of Finite-state Concurrent Systems, *Ann. Rev. Comput.Sci.*, 1987.2,260-290, 1987.

[Clarke 89] E.M. Clarke, E.E. Long, and K.L. McMillan, Compositional Model Checking, *Proc. 4th IEEE Symp. on Logic in Computer Science (LICS)*, 1989.

[Cleaveland 93] R. Cleaveland, J. Parrow, B. Steffen, The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems, *ACM Trans. on Programming Languages and Systems*, Vol.15, No.1, 1993.

[CAV 89] J. Sifakis, (Ed.), *Automatic Verification Methods for Finite State Systems. Lecture Notes in Computer Science*, Vol.407, Springer-Verlag, 1989.

[CAV 93] C. Courcoubetis (Ed.), *CAV'93 Conference on Computer-Aided Verification, Lecture Notes in Computer Science*, Vol.697, Springer-Verlag, 1993.

[CAV 94] D. L. Dill (Ed.), *CAV'94 Conference on Computer-Aided Verification, Lecture Notes in Computer Science*, Vol.818, Springer-Verlag, 1994.

[Conway 71] J. H. Conway, *Regular Algebra and Finite Machines*, Chapman and Hall, 1971.

[Davis 90] A.M. Davis, *Software Requirements Analysis & Specification*, Prentice Hall, 1990.

[Emerson 82] E.A. Emerson and E.M. Clarke, Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming*, **2**, 1982.

[Emerson 85a] E.A. Emerson and J.Y. Halpern, Decision Procedures and Expressiveness in the Temporal Logic of Branching Time, *J. Computer and System Sciences*, **30**, 1985.

[Emerson 85b] E.A. Emerson, C.-L. Lei, Modalities for Model Checking: Branching Time Strikes Back, *Proc. 12th ACM Symp. on Principles of Programming Languages (POPL)*, 1985.

[Emerson 90a] E.A. Emerson, Temporal and Modal Logic, in *Handbook of Theoretical Computer Science*, Volume B, Formal Methods and Semantics, The MIT Press, 1990.

[Emerson 90b] E.A. Emerson and J. Srinivasan, A Decidable Temporal Logic to Reason about Many Processes, *Proc. ACM 9th Symp. on Principle of Distributed Computing (PODC)*, 1990.

[Etessami 91] F.S. Etessami and G.S. Hura, Rule-Based Design Methodology for Solving Control Problems, *IEEE Trans. on Software Engineering*, Vol.17, No.3, 1991.

[Fantechi 91] A. Fantechi, S. Gnesi, G. Ristori, Compositionality and bisimulation: A negative result, Information Processing Letters, 39, 109–114, 1991.

[Feldbrugge90] F.Feldbrugge : “Petri Net Tool Overview 1989”, *Advances in Petri Nets 1989*, *Lecture Notes in Computer Science*, Vol.424, Springer-Verlag, 1990.

[Felder 94] M. Felder, D. Mandrioli, A. Morzenti, Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models, *IEEE Trans. on Software Engineering*, Vol.20, No.2, 1994.

[FGCS 92] MENDELS ZONE, Demonstration at *Internat. Conf. Fifth Generation Computer Systems (FGCS'92)*, 1992.

[Fukuoka 91] K. Fukuoka, A. Yokozawa, K. Tamaru, Hierarchical design of a μITRON specification kernel: TR2, *Proc. The Eighth TRON Project Symposium*, IEEE Comput. Soc. Press, 1991.

[Furukawa 92] K. Furukawa, Logic programming as the integrator of the Fifth Generation Computer Systems project, *Commun. ACM*, Vol.35, No.3, 82-92, 1992.

[Gabrielian 91] A. Gabrielian and M.K. Franklin, Multi-level Specification and Verification of Real-Time Software, *Proc. 12th Internat. Conf. on Software Engineering (ICSE)*, 1990.

[Galton 81] A. Galton, Temporal Logic and Computer Science: An Overview, *Temporal Logics and Their Applications (A. Galton, ed.)*, Academic Press, 1 – 52, 1987.

[Genrich 81] H.J. Genrich and K. Lautenbach, System Modeling with High-Level Petri Nets, *Theoretical Computer Science*, **13**, 1981.

[Ghezzi 93] C. Ghezzi, H. Felder, M. Paul, Real-Time Systems: A Survey of Approaches to Formal Specification and Verification, *Proc. on 4th European Software Engineering Conference (ESEC)*, 1993.

[Giovanni91] R. Di Diovanni, Hood Nets, *Advances in Petri Nets 1991*, *Lecture Notes in Computer Science*, Vol.524, Springer-Verlag, 1991.

[Godefroid 91a] P. Godefroid, P. Wolper, A Partial Approach to Model Checking, *Proc. 6th IEEE Symp. on Logic in Computer Science (LICS)*, 1991.

[Godefroid 91b] P. Godefroid, and P. Wolper, Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties, *Conference on Computer-Aided Verification (CAV'91)*, *Lecture Notes in Computer Science*, Vol.575, Springer-Verlag, 1991, also *Formal methods in System Design*, Vol.2, No.2, Kluwer Academic Publishers, 1993.

[Godefroid 96] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem —, *Lecture Notes in Computer Science*, Vol.1032, Springer-Verlag, 1996.

[Gomaa 93] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley , 1993.

[Kawata 95] H. Kawata and N. Uchihira, Automatic Generation of Plant Simulator to Verify Sequence Control Software (in Japanese), *SICE Federated Symposium on Systems and Information*, Toyama, Nov., 1995.

[Kawata 96] H. Kawata and N. Uchihira, Practical Program Validation for Plant Control Systems Using SFC and Temporal Logic, *1996 IEEE International Conference on Systems, Man, and Cybernetics (SMC'96)*, 1996.

[Katai 82] O. Katai and S. Iwai, Construction of Scheduling Rules for Asynchronous, Concurrent Systems Based on Tense Logic (in Japanese), *Trans. of SICE*, Vol. 18, No. 12, 1982.

[Kojima 91] F. Kojima and T. Koike, Advanced Configuration Tolls for DCS, *ISA Trans.*, Vol.30, No.2, 1991.

[Kanellakis 90] P. C. Kanellakis, S. A. Smolka, CCS Expressions, Finite State Processes and Three Problems of Equivalence, *Information and Computation*, **86**, 1990.

[Koymans 87] R. Koymans, Specifying Message Passing Systems Requires Extending Temporal Logic, in *Temporal Logic in Specification* (B. Banieqbal, H. Barringer, A. Pnueli, eds.) *Lecture Notes in Computer Science*, Vol.398, Springer-Verlag, 1987.

[Kröger 87] F. Kröger, *Temporal Logic of Programs*, Springer-Verlag, 1987.

[Hale 87] R. Hale, Using Temporal Logic for Prototyping: The Design of a Lift Controller, in *Temporal Logic in Specification* (B. Banieqbal, H. Barringer, A. Pnueli, eds.) *Lecture Notes in Computer Science*, Vol.398, Springer-Verlag, 1987.

[Harel 87a] D. Harel, Statechart: A visual formalism for complex systems, *Sci. Comput. Program.*, Vol.8, No.3, 1987.

[Harel 87b] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman, On the Formal Semantics of Statechart, *Proc. 2nd IEEE Symp. on Logic in Computer Science (LICS)*, 1987.

[Harel 90] D. Harel, et al., STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. on Software Engineering*, Vol.16, No.4, 1990.

[Hatley 87] D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.

[Hennessy 85a] M. Hennessy, and R. Milner, Algebraic Laws for Nondeterminism and Concurrency, *Journal of ACM*, Vol. 32, No. 1, 1985.

[Hennessy 85b] M. Hennessy and C. Stirling, The Power of the Future Perfect in Program Logics, *Information and Control*, **67**, 1985.

[Hennessy 88] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

[Henzinger 91] T. A. Henzinger, Z. Manna, A. Pnueli, Timed Transition Systems, *Real-Time: Theory and Practice*, *Lecture Notes in Computer Science*, Vol.600, Springer-Verlag, 1991.

[Hiraishi 95] K. Hiraishi and M. Nakano, On Symbolic Model Checking in Petri Nets, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E78-A, No.11, 1995.

[Hoare 84] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1984.

[Holzmann 91] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[Honiden 86a] S. Honiden, N. Uchihira, T. Kasuya, MENDEL: Prolog Based Concurrent Object Oriented Language, *Proc. IEEE COMPCON'86*, 1986.

[Honiden 86b] S. Honiden. N. Uchihira, A. Ohsuga, and T. Kasuya, MENDEL: Meta-Inferential System Description Language (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol. 27, No. 2, 219–217, 1986.

[Honiden 90] S. Honiden, N. Uchihira, K. Matsumoto, K. Matsumura, M. Arai, An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, *Real-Time Systems*, Vol.1, No.4, 313–331, 1990.

[Honiden 91a] S. Honiden, A. Ohsuga, N. Uchihira, An Integration Environment to Put Formal Specifications into Practical Use in Real-Time Systems, Proc. 6th IWSSD, 1991.

[Honiden 91b] S. Honiden, N. Uchihira, K. Itoh, An Application of Artificial Intelligence to Prototyping Process in Performance Design for Real-Time Systems, *Proc. on 3rd European Software Engineering Conference (ESEC), Lecture Notes in Computer Science*, Vol.550, Springer-Verlag, 1991.

[Honiden 92] S. Honiden and N. Uchihira, Net-Oriented Analysis and Design, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.10, 1992.

[Honiden 94] S. Honiden, K. Nishimura, N. Uchihira, K. Itoh, An Application of Artificial Intelligence to Object-Oriented Performance Design for Real-Time Systems, *IEEE Trans. on Software Engineering*, Vol.20, No.11, 1994.

[Honiden 96] S. Honiden, A. Ohsuga, N. Uchihira, MENDELS ZONE: A parallel program development system based on formal specifications, *Information and Software Technology*, Vol.38, 1996.

[Howell 88] R.R. Howell, L.E. Rosier, and H.C. Yen, A Taxonomy of Fairness and Temporal Logic Problems for Petri Nets, *Lecture Notes in Computer Science*, Vol.324, Springer-Verlag, 1988.

[Huber 90] P. Huber, et al., Hierarchies in Coloured Petri Nets, Advances in Petri Nets 1990, *Lecture Notes in Computer Science*, Vol.483, Springer-Verlag, 1990.

[Ichikawa 85] A. Ichikawa, K. Yokoyama, and S. Kurogi, Reachability and Control of Discrete Event Systems Represented by Conflict-Free Petri Nets, *Proc. IEEE Internat. Symp. on Circuits and Systems*, 1985.

[IEC 1131-3] IEC 1131-3, IEC International Standard for Programmable Controllers, Part 3: Programming Languages (IEC 1131-3), International Electrotechnical Commission, 1993.

[IWSSD 87] Problem Set, in Proceedings of *the 4th International Workshop on Software Specification and Design (IWSSD)*, 1987.

[Jensen 90] K. Jensen, Coloured Petri Nets: A High Level Language for System Design and Analysis, *Advances in Petri Nets 1990*, *Lecture Notes in Computer Science*, Vol.483, Springer-Verlag, 1990.

[Jensen 92] K. Jensen, *Coloured Petri Nets, Basic Concept, Analysis Methods and Practical Use*, Volume 1, Springer-Verlag, 1992.

[Jensen 95] K. Jensen, *Coloured Petri Nets, Basic Concept, Analysis Methods and Practical Use*, Volume 2, Springer-Verlag, 1995.

[Josko 87] B. Josko, MCTL – An Extension of CTL for Modular Verification of Concurrent Systems, in *Temporal Logic in Specification* (B. Banieqbal, H. Barringer, A. Pnueli, eds.) *Lecture Notes in Computer Science*, Vol.398, Springer-Verlag, 1987.

[Lakos 95] C. Lakos, From Coloured Petri Nets to Object Petri Nets, *Proc. 16th Internat. Conf. on Application and Theory of Petri Nets (ICATPN), Lecture Notes in Computer Science*, Vol.935, Springer-Verlag, 1995.

[Lamport 94] L. Lamport, The Temporal Logic of Actions, *ACM Trans. on Prog. Lamg. Syst.*, Vol.16, No.3, 1994.

[Lee 91] E. S. Lee, et al., Construction and Implementation of a Specification Environment SEGL Based on G-LOTOS (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol.32, No.3, 1991.

[Lee 85] K. Lee and J. Favrel, Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets, *IEEE Trans. on SMC,* Vol.SMC-15, No.2, 1985.

[Lynch 86] N. Lynch and M. Merritt, Introduction to the Theory of Nested Transactions, *Proc. Internat. Conf. on database Theory (ICDT'86)*, 1986.

[Lynch 88] N. Lynch and M. Tuttle, Introduction to Input/Output Automata, *MIT Technical Report MIT/LCS/TM-373*, 1988.

[Marsan 86] M. Ajmone Marsan, G.Balbo and G.Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, 1986.

[Marsan 95] M. Ajmone Marsan, G. Balbo and G. Conte, S. Donatelli, and G. Franceschinis, *Modeling with Generalized Stochastic Petri Nets*, John Wiley & Sons, 1995.

[McMillan 93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[Manna 81a] Z. Manna and A. Pnueli, Verification of Concurrent Programs, Part I: The Temporal Framework, *Stanford University Technical Report*, No.STAN-CS-81-836, 1981.

[Manna 81b] Z. Manna and A. Pnueli, Verification of Concurrent Programs, Part II: Temporal Proof Principles, *Stanford University Technical Report*, No.STAN-CS-81-843, 1981.

[Manna 84] Z.Manna and P.Wolper, Synthesis of Communicating Processes from Temporal Logic Specification, *ACM Trans. Program. Lang. & Syst.*, Vol. 6, No. 1, pages 68 - 93, 1984.

[Manna 92] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems - specification -, Springer-Verlag , 1992.

[Milner 81] R. Milner, A Modal Characterization of Observable Machine-behaviour, *Lecture Notes in Computer Science*, Vol.112, Springer-Verlag, 1981.

[Milner 89] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[Mishra 85] B. Mishra, and E.M. Clarke, Hierarchical Verification of Asynchronous Circuit Using Temporal Logic. *Theoretical Computer Science*, **38**, 1985.

[Miyagi 88] P.E.Miyagi, K.Hasegawa, K.Takahashi, A Programming Language for Discrete Event Production Systems Based on Production Flow Schema and Mark Flow Graph, Trans. of the Society of Instrument and Control Engineering, Vol.24, No.2, Feb. 1988.

[Moszkowski 86] B.C. Moszkowski, *Executing Temporal Logic Programs*, Cambridge Univ. Press, 1986.

[Murata 89] T. Murata, Petri Nets: Properties, Analysis and Applications, *Proc. IEEE*, Vol. 77, No. 4, 1989.

[Murata 90] T. Murata and N. Komoda, Real-Time Control Software for Transaction Processing Based on Colored Safe Petri Net Model, *Real-Time Systems*, Vol.1, No.4, 299–312, 1990.

[Nagao 92] Y. Nagao, et al., Petri Net Based Programming System for FMS, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.10, 1992.

[Ostroff 90] J.S. Ostroff, *Temporal Logics for Real-Time Systems* Research Studies Press, 1990.

[Paige 87] R. Paige, R.E.Tarjan, Three Partition Refinement Algorithms, *SIAM J. Comput.*, 16, No.6, 1987.

[Park 81] D.Park, Concurrency and automata on infinite sequences, *Lecture Notes in Computer Science*, Vol.104, Springer-Verlag, 1981.

[Patent KOUKAI H4-259071] M. Arami, N. Uchihira, Graph Generation Method, *Japan Patent Office KOUKAI H4-259071*, 1991.

[Peterson 81] J.L.Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc., 1981.

[Pinci 91] V.O.Pinci and R.M.Shapiro, An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets, Advances in Petri Nets 1991, *Lecture Notes in Computer Science*, Vol.524, Springer-Verlag, Springer-Verlag , 1991.

[Pinter 84] S.S. Pinter, P. Wolper, A Temporal Logic for Reasoning about Partially Ordered Computations, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984.

[Plaisted 86] D. A. Plaisted, A Decision Procedure for Combinations of Propositional Temporal Logic and Other Specialized Theories, *Journal of Automated Reasoning*, **2**, 1986.

[Pnueli 77] A. Pnueli, The Temporal Logic of Programs, *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977.

[Pnueli 81] A. Pnueli, Temporal Semantics of Concurrent Programs, *Theoretical Computer Science*, **13**, 1981.

[Pnueli 86] A. Pnueli, Application of Temporal Logic To the Specification and Verification of Reactive Systems: A survey of Current Trends, in *Current Trends in Concurrency* (J.W. de Backker, W.-P. de Roever, G. Rozenverg, eds.), *Lecture Notes in Computer Science*, Vol.224, Springer-Verlag, 1986.

[Pnueli 89a] A. Pnueli and R. Rosner, On the Synthesis of an Asynchronous Reactive Module, *16th International Colloquium on Automata, Languages, and Programming (ICALP)*, *Lecture Notes in Computer Science*, Vol.372, Springer-Verlag, 1989.

[Pnueli 89b] A. Pnueli and R. Rosner, On the synthesis of a reactive module, *Proc. 16th ACM Principle of Programming Languages (POPL)*, 1989.

[Pnueli 90] A. Pnueli and R. Rosner, Distributed Reactive Systems are Hard to Synthesize, *Proc. IEEE 31st Annual Symp. on Foundations of Computer Science (FOCS)*, 1990.

[Ramadge 89] P.J. Ramadge and W.M. Wonham, The Control of Discrete Event Systems, *Proc. IEEE*, Vol.77, No.1, 81–98, 1989.

[Rescher 71] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, 1971.

[Reisig 87] W. Reisig, Petri Nets in Software Engineering, *Advances in Petri Nets*, *Lecture Notes in Computer Science*, Vol.255, Springer-Verlag, 63–96, 1987.

[Reisig 91] W. Reisig, Petri Nets and Algebraic Specification *Theoretical Computer Science*, **80**, 1991.

[Reisig 92] W.Reisig, *A Primer in Petri Net Design*, Springer-Verlag,1992.

[Ross 77] D.T. Ross, Structured Analysis: A Language for Communicating Idea, *IEEE Trans. on Software Engineering*, Vol.3, No.1, 1977.

[Rumbaugh 91] J.Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Saeki 87] M. Saeki, Non-Monotonic Propositional Temporal Logic and its Application to Formal Specifications (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol.28, No.6, 1987.

[Shatz 93] S. M. Shatz, *Development of Distributed Software: Concepts and Tools,* Macmillan, New York, 1993.

[Sistla 84] A.P. Sistla, E.M. Clarke, N. Francez, A.R. Meyer, Can Message Buffers Be Axiomatized in Linear Temporal Logic?, *Information and Control*, **63**, 1984.

[Stankovic 95] J.A. Stankovic, Implications of Classical Scheduling Results for Real-Time Systems, *IEEE Computer*, Vol.28, No.6, 1995.

[Stirling 87] C. Stirling, Modal Logic for Communicating Systems, *Theoretical Computer Science*, **49**, 1987.

[Stirling 89a] C. Stirling, Temporal Logic for CCS, *Lecture Notes in Computer Science*, Vol.354, Springer-Verlag, 1989.

[Stirling 89b] C. Stirling and D. Walker, CCS, Liveness, and Local Model Checking in Linear Time Mu-calculus, *Lecture Notes in Computer Science*, Vol.407, Springer-Verlag, 1989.

[Suzaki 91] K.Suzaki and T.Chikayama, AYA: Process-Oriented Concurrent Programming Language on KL1 (in Japanese), *Proc. KL1 Programming Workshop'91*, 1991.

[Suzuki 89] I. Suzuki and H. Lu, Temporal Petri Nets and Their Application to Modeling and Analysis of a Handshake Daisy Chain Arbiter, *IEEE Trans. on Computers*, Vol.38, No.5, 1989.

[Suzuki 90] I. Suzuki, Formal Analysis of the Alternating Bit Protocol by Temporal Petri Nets, *IEEE Trans. on Software Engineering*, Vol.16, No.11, 1990.

[Taki 89] K. Taki, The FGCS Computing Architecture, *Information Processing*, **89**, *(Proc. IFIP 11th World Computer Congress)*, 627–632, 1989.

[Leeuwen 90] J. Van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Volme B, Formal Models and Semantics, The MIT Press/Elsevier, 1990.

[Uchihira 87] N. Uchihira, T. Kasuya, K. Matsumoto, S. Honiden, Concurrent Program Synthesis with Reusable Components Using Temporal Logic, Proc. IEEE COMPSAC'87, 1987.

[Uchihira 88] N. Uchihira, K. Matsumoto, S. Honiden, H. Nakamura, MENDELS: Concurrent Program Synthesis System Using Temporal Logic, *Proc. 6th Logic Programming Conference (K. Furukawa, et al., eds.)*, *Lecture Notes in Computer Science*, Vol.327, Springer-Verlag, 50–68, 1988.

[Uchihira 89a] N. Uchihira, K. Nishimura, S. Sumida, and H. Kawata. Verification and Debugging of Concurrent Robot Control Programs Using Temporal Logic (in Japanese). *The 3rd National Meeting of Japan Society of Artificial Intelligence*, 1989.

[Uchihira 89b] N. Uchihira and S. Honiden, Concurrent Programming Language Based on Petri Nets on Intelligent Distributed Processing System (in Japanese). *IEICE Technical Report CPSY 89-34*, 1989.

[Uchihira 90a] N. Uchihira, H. Kawata, K. Matsumoto, M. Ito, S. Honiden, Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, *Proc. IEEE 23rd Hawaii International Conference on System Science (HICSS)*, 1990.

[Uchihira 90b] N. Uchihira and S. Honiden, Verification and synthesis of concurrent programs using Petri nets and temporal logic, *Trans. on IEICE*, Vol.E73, No.12 , 1990.

[Uchihira 92a] N. Uchihira, PQL: Modal Logic for Compositional Verification of Concurrent Programs (in Japanese), *Trans. IEICE* Vol.J75-DI, No.2 , 1992), also its english version, *Systems and Computers in JAPAN*, Vol. 25, No.1, Jan., Scripta Technica (John Wiley & Sons), 1994.

[Uchihira 92c] N. Uchihira, Compositional synthesis for cooperating discrete event systems from modular temporal logic specifications, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.3, 1992.

[Uchihira 92b] N. Uchihira, M. Arami, S. Honiden, A Petri-Net-Based Programming Environment and Its Design Methodology for Cooperating Discrete Event Systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.10, 1992.

[Uchihira 92d] N. Uchihira and S. Honiden, Petri Net and Temporal Logic (in Japanese), *Petri nets and its Application*, 240–251, SICE Publishing Office, 1992.

[Uchihira 92e] N. Uchihira and S. Honiden, MENDELS ZONE: Petri-net-based Programming Environment for Cooperative Discrete Event Systems (in Japanese), *SICE 10th Workshop on Discrete Event Systems*, 21–28, Niigata, 1992.

[Uchihira 93a] N. Uchihira, M. Arami, H. Kawata, Program Verification for Sequence Control Systems, *Toshiba Review*, Vol.48, No.10, 1993.

[Uchihira 93b] N. Uchihira and S. Honiden, Software Design Methodology using High-Level Petri Nets (in Japanese), *SICE 12th Workshop on Discrete Event Systems*, 73-80, Matsuyama, 1993.

[Uchihira 95a] N. Uchihira, S. Honiden, Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *IEEE 28th Hawaii International Conference on System Science (HICSS)*, 1995, also in *J. Systems and Software*, Vol.33, No.3, 207-221, 1996.

[Uchihira 95b] N. Uchihira and H. Kawata, Practical Program Validation for State-Based Reactive Concurrent Systems – Harmonization of Simulation and Verification –, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E78-A, No.11, 1995.

[Uchihira 96a] N. Uchihira, MENDELS ZONE, Tool Presentation, *Proc. 17th Internat. Conf. on Application and Theory of Petri Nets (ICATPN)*, Osaka, 1996.

[Uchihira 96b] N. Uchihira and S. Honiden, A High-Level Petri Net for Accurate Modeling of Reactive and Concurrent Systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E79-A, No.11, 1996.

[Uchihira 96c] N. Uchihira, S. Honiden, T. Seki, Hypersequential Programming – A Novel Paradigm for Concurrent Programming –, *1st International Workshop on Software Engineering for Parallel and Distributed Systems*, Berlin, 1996.

[Uchihira 97a] N. Uchihira and S. Honiden, Three Phase Net-Oriented Software Design Method (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol.38, No.1, 1997.

[Uchihira 97b] N. Uchihira, H. Kawata, Scenario-Based Hypersequential Programming: Concept and Example, *2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, Boston, 1997.

[Uchihira 97c] N. Uchihira, S. Honiden, T. Seki, Hypersequential Programming — A New Paradigm for Concurrent Program Development —, *IEEE Concurrency*, Vol.5, No.3, 1997.

[Ueda 90] K. Ueda and T. Chikayama, Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, Vol.33, No.6, 494–500 (1990).

[Uraoka 92] T. Uraoka, J. Yamamoto, A. Ohsuga, S. Honiden, An Algebraic Specification and Verification of a Plant Control Expert System (in Japanese), *IPSJ Technical Report, SE 86-15*, 1992.

[Valk 83] R.Valk, Infinite Behavior of Petri Nets, *Theoretical Computer Science*, **25**, 1983.

[Valk 85] R.Valk and M.Jantzen, The Residue of Vector Sets with Applications to Decidability Problems in Petri Nets, *Acta Informatica*, **21**, 1985.

[Vallejo 94] F. Vellejo, J.A. Gregorio, M. G. Harbour, J.M. Drake, Shared Memory Multimicroprocessor Operating System with an Extended Petri Net Model, *IEEE Trans. on Parallel Distrib. Syst.*, Vol.5, No.7, 1994.

[Valmari 90] A. Valmari, Stubborn Sets for Reduced State Space Generation, *Proc. 11th Internat. Conf. on Application and Theory of Petri Nets (ICATPN), Lecture Notes in Computer Science*, Vol.483, Springer-Verlag, 491–515, 1990.

[Vardi 86] M.Y. Vardi and P. Wolper, An Automata-Theoretic Approach to Automatic Program Verification, *Proc. 1st IEEE Symp. on Logic in Computer Science (LICS)*, 1986.

[Walker 90] D.J. Walker, Bisimulation and Divergence. *Information and Computation*, **85**, 1990.

[Ward 85] P. Ward and S. Mellor, *Structured Development for Real-time Systems*, Vol.1 – Vol.4, Prentice Hall, 1985.

[Ward 86] P. Ward, The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Trans. on Software Engineering*, Vol.12, No.2, 1986.

[Winskel 90] G. Winskel, On the Compositional Checking of Validity, *Proc. 1st Internat. Conf. on Concurrency Theory (CONCUR'90)*, *Lecture Notes in Computer Science*, Vol.458, Springer-Verlag, 1990.

[Wolper 83a] P. Wolper, M.Y. Vardi, and A.P. Sistla, Reasoning about Infinite Computation Paths, *Proc. IEEE 24th Annual Symp. on Foundations of Computer Science (FOCS)*, 1983.

[Wolper 83b] P. Wolper, Temporal Logic Can Be More Expressive, *Information and Control*, **56**, 1983.

[Wolper 88] P. Wolper, On the Relation of Programs and Computations to Models of Temporal Logic, *Temporal Logic in Specification (B. Banieqbal, H. Barringer, A. Pnueli, eds.)*, *Lecture Notes in Computer Science*, Vol.398, Springer-Verlag, 1987.

[Wolper 93] P. Wolper and P. Godefroid, Partial-Order Methods for Temporal Verification, *Proc. 4th Internat. Conf. on Concurrency Theory (CONCUR'93)*,*Lecture Notes in Computer Science*, Vol.715, Springer-Verlag, 1993.

[Yoshida 88] K. Yoshida and T. Chikayama, A'UM – Stream-Based Concurrent Object-Oriented Language –, *Proc. Internat. Conf. on Fifth Generation Computer Systems 1988 (FGCS88)*, ICOT, 1988.

[Yoshida 90] K. Yoshida, *A'UM* A Stream-Based Concurrent Object-Oriented Programming Language, *Ph.D Thesis*, Keio University, 1990.

[Yonezaki 91] N. Yonezaki, Conceptual Modeling in MSL, *Advances in Information Modeling and Knowledge Bases*, IOS Press, 1991.

[Yoneda 93] T. Yoneda, A. Shibayama, B-H. Schlingloff, E.M. Clarke, Efficient Verification of Parallel Real-Time Systems, *Conference on Computer-Aided Verification (CAV'93)*, *Lecture Notes in Computer Science*, Vol.697, Springer-Verlag, 1993.

[Yoshimura 93] N. Yoshimura, N. Yonezaki, More Expressive Temporal Logic for Specification, *5th Internat. Conf. on Software Engineering and Knowledge Engineering*, 1993.

[Zave 82] P. Zave, An Operational Approach to Requirements Specification for Embedded Systems, *IEEE Trans. on Software Engineering*, Vol.8, No.3, 1982.

[Zave 91] P. Zave, An Insider's Evaluation of PAISLey, *IEEE Trans. on Software Engineering*, Vol.17, No.3, 1991.

[Zhou 92] M.C. Zhou, F. DiCesare, A. Desrochers, Hybrid Methodology for Synthesis of Petri Net Models for Manufacturing Systems, *IEEE Trans. Robotics and Automation*, Vol.8, No.3 , 1992.

Publications by the Author

Journals

1. N. Uchihira and S. Honiden, Verification and synthesis of concurrent programs using Petri nets and temporal logic, *Trans. IEICE*, Vol.E73, No.12, 2001-2010, 1990.
2. N. Uchihira, PQL: Modal Logic for Compositional Verification of Concurrent Programs (in Japanese), *Trans. IEICE*, Vol.J75-DI, No.2, 76-87, 1992, also its English version, *Systems and Computers in JAPAN*, Vol. 25, No.1, Scripta Technica (John Wiley & Sons), 1994.
3. N. Uchihira, Compositional Synthesis for Cooperating Discrete Event Systems from Modular Temporal Logic Specifications, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.3, 380-391, March, 1992.
4. N. Uchihira, M. Arami, S. Honiden, A Petri-Net-Based Programming Environment and its Design Methodology for Cooperating Discrete Event Systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.10, 1335-1347, 1992.
5. N.Uchihira and H. Kawata, Practical Program Validation for State-Based Reactive Concurrent Systems – Harmonization of Simulation and Verification –, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E78-A, No.11, 1995.
6. N. Uchihira and S. Honiden, Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *J. Systems and Software*, Vol.33, No.3, 207-221, 1996.
7. N. Uchihira and S. Honiden, A High-Level Petri Net for Accurate Modeling of Reactive and Concurrent Systems, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E79-A, No.11, 1797-1808, 1996.
8. N. Uchihira and S. Honiden, Three Phase Net-Oriented Software Design Method (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol.38, No.1, 1997.
9. N. Uchihira, S. Honiden, T. Seki, Hypersequential Programming — A New Paradigm for Concurrent Program Development —, *IEEE Concurrency*, Vol.5, No.3, 44-54, 1997.
10. S. Honiden. N. Uchihira, A. Ohsuga, and T. Kasuya, MENDEL: Meta-Inferential System Description Language (in Japanese), *Trans. Inf. Process. Soc. Japan*, Vol. 27, No. 2, 219-217, 1986.
11. S. Honiden, N. Uchihira, K. Matsumoto, K. Matsumura, M. Arai, An Application of Structural Modeling and Automated Reasoning to Real-Time Systems Design, *Real-Time Systems*, Vol.1, No.4, 313-331, 1990.
12. S. Honiden, N. Sueda, A. Hoshi, N. Uchihira, K. Mikame, Software Prototyping with Reusable Components, *J. Inf. Process. Japan*, Vol. 9, No. 3, 123-129, 1986.
13. S. Honiden, A. Ohsuga, and N. Uchihira, An integration method of real time SA and object oriented design using algebraic and temporal logic specifications (in Japanese) *Trans. Inf. Process. Soc. Japan*, Vol.33, No.2, 173-182, 1992.

14. S. Honiden and N. Uchihira, Net-oriented analysis and design, *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, Vol.E75-A, No.10, 1317-1325, 1992.

15. S. Honiden, K. Nishimura, N. Uchihira, K. Itoh, An Application of Artificial Intelligence to Object-Oriented Performance Design for Real-Time Systems, *IEEE Trans. on Software Engineering*, Vol.20, No.11, 1994.

16. S. Honiden, A. Ohsuga, N. Uchihira, MENDELS ZONE: A parallel program development system based on formal specifications, *Information and Software Technology*, Vol.38, 181–189, 1996.

Conferences

1. N. Uchihira, T. Kasuya, K. Matsumoto, S. Honiden, Concurrent Program Synthesis with Reusable Components Using Temporal Logic, *Proc. IEEE COMPSAC'87*, Tokyo, 1987.

2. N. Uchihira, K. Matsumoto, S. Honiden, and H. Nakamura, MENDELS: Concurrent Program Synthesis System Using Temporal Logic, *Proc. the 6th Logic Programming Conference*, Tokyo, 1987, also in *Lecture Notes in Computer Science*, Vol.327, Springer-Verlag, 1988.

3. N. Uchihira, H. Kawata, K. Mastumoto, M. Ito, S. Honiden, Synthesis of Concurrent Programs: Automated Reasoning Complements Software Reuse, *23rd Hawaii International Conference on System Science (HICSS)*, 1990.

4. N. Uchihira and S. Honiden, Compositional Adjustment of Concurrent Programs to Satisfy Temporal Logic Constraints in MENDELS ZONE, *Proc. IEEE 28th Hawaii International Conference on System Science (HICSS)*, 1995.

5. N. Uchihira, S. Honiden, T. Seki, Hypersequential Programming – A Novel Paradigm for Concurrent Programming –, *1st International Workshop on Software Engineering for Parallel and Distributed Systems*, Berlin, Chapman & Hall, 1996.

6. N. Uchihira, H. Kawata, Scenario-Based Hypersequential Programming: Concept and Example, *2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, Boston, IEEE Computer Society Press, 277-283, 1997.

7. N. Uchihira, H. Kawata, F. Tamura, Scenario-Based Hypersequential Programming: Formulation of Parallelization, *International Symposium on High Performance Computing (ISHPC)*, Fukuoka, *Lecture Notes in Computer Science*, Vol.1336, Springer-Verlag, 267-240, 1997.

8. N. Uchihira, How to Make Concurrent Programs Highly Reliable – More Than State Space Analysis, *International Conference on Application of Concurrency to System Design*, Aizu-Wakamatsu, IEEE Computer Society Press, 16-23, 1998.

9. S. Honiden, N. Uchihira, K. Mastumoto, K. Matsumura, M. Arai, An Application of Structural Modeling and Automated Reasoning to Concurrent Program Design, *Proc. IEEE 22th Hawaii International Conference on System Science (HICSS)*, Vol.II: Software Track, 134-141, 1989.

10. S. Honiden, A. Ohsuga, and N. Uchihira, An Integration Environment to Put Formal Specifications into Practical Use in Real-time Systems, *Proc. the Sixth International Workshop on Software Specification and Design (IWSSD)*, IEEE Comput. Soc. Press, 102-109, 1991.

11. S. Honiden, N. Uchihira, and K. Itoh, An Application of Artificial Intelligence to Prototyping Process in Performance Design for Real-time Systems, *Proc. 3rd European Software Engineering Conference (ESEC)*, *Lecture Notes in Computer Science*, Vol.550, Springer-Verlag, 1991.

12. S. Honiden, N. Uchihira, K. Matsumoto, and K. Itoh, A Prototyping Process for Performance Design in Real-time Systems, *Proc. InfoJapan '90*, Vol.1, 103-110, North-Holland, 1990.

13. H. Kawata and N.Uchihira, Practival Program Validation for Plant Control Systems Using SFC and Temporal Logic, *1996 IEEE International Conference on Systems, Man, and Cybernetics (SMC'96)*, 1996.

Articles

1. S. Honiden, N. Uchihira, K. Matsumoto, Temporal Logic and Petri Nets (in Japanese), *Operations Research*, Japan Society of Operations Research, Oct., 1987.
2. S. Honiden, N. Uchihira, H. Nakamura, Automatic Programming for Control Systems (in Japanese), *Inf. Process. Soc. Japan*, Vol.28, No.10, 1398-1404, 1987.
3. K. Matsumoto, N. Uchihira, S. Honiden, Temporal Logic and their Applications (in Japanese), *Inf. Process. Soc. Japan*, Vol.30, No.6, 651-657, 1989.
4. N. Uchihira, S. Honiden, Petri Nets and Temporal Logic (in Japanese), SICE Special Interest Group for Discrete Event Systems (ed.), *Petri Nets and Its Applications*, SICE, 1992.
5. M. Aoyama, K. Hiraishi, N. Uchihira, Software Development Methodologies Based on High Level Petri Nets (in Japanese), *Computer Software*, Vol.11, No.4, 3-19, Japan Society for Software Science and Technology (JSSST), 1994.
6. N. Uchihira and H. Kawata, Exhaustive Simulation of Discrete Event Systems (in Japanese), *Journal of The Society of Instrument and Control Engineers (SICE)*, Vol.35, No.10, 763-769, 1996.
7. N. Uchihira and H. Kawata, Testing Control Programs: A Survay (in Japanese), *Journal of IPSJ* Vol.39, No.1, 19-25, 1998.
8. N. Uchihira, K. Hiraishi, M. Aoyama, Petri Nets: Book and Tool Review (in Japanese), *Journal of IPSJ* Vol.39, No.1, 67-70, 1998.

Books

1. K. Itoh, S. Honiden, N. Uchihira, *Prototyping Tools* (in Japanese), Keigaku Shuppan, 1987.
2. M. Aoyama, N. Uchihira, K. Hiraishi, *Petri Nets – Theory and Practice –* (in Japanese), Asakura Shoten, 1995.

Index