〔非公開〕

ＴＲ－Ｃ－０１３６

Automated Generation of Communication Software from Service Specifications

田倉　昭
Akira TAKURA

１９９６　３．７

ＡＴＲ通信システム研究所

Automated Generation of Communication Software from Service Specifications

Akira Takura

March 7, 1996

ATR Communication Systems Research Laboratories

Acknowledgments

I am extremely grateful to Associate Prof. Etsuya Shibayama for his valuable comments on this thesis. I am also very grateful to Associate Prof. Masako Horai for her valuable comments. I wish to express my sincere gratitude to Prof. Kojiro Kobayashi, Prof. Masataka Sassa and Associate Prof. Motoshi Saeki for their willingness to referee this Ph. D. thesis and for their valuable comments.

I thank Dr. Kohei Habara, Dr. Nobuyoshi Terashima and Dr. Tadashi Ohta for their encouragements concerning this work while I have been at ATR. In particular, Dr. Ohta offered many valuable comments and helpful discussion on the work. I am also greatly indebted to my colleagues at ATR Communication Systems Research Laboratories, especially Keizo Kawata, Kenji Shibata, Mitsuhiro Nakamura, Yoshihiro Ueda, Takafumi Sera, Yoshiki Kondo, Tsuneki Haizuka, Kazumasa Takami and Yasuro Kawarasaki who collaborated with me in making the communication software generation system that is referred to in this thesis. I also thank them for their helpful discussions.

I would also like to thank the many people who helped me at NTT, especially Dr. Haruhisa Ichikawa, Dr. Yutaka Hirakawa, who introduced me to STR, and Atsushi Kanai, and Dr. Keiji Okada at NTT Advanced Technology.

Contents

Chapter 1

Introduction

1.1 Automated Generation of Communication Software

Automated software generation is a promising and efficient way of developing reliable software. This research aims at automating communication software generation from formally described requirement specifications.

In communication systems services are added frequently. For this reason, the ability to develop communication software efficiently and reliably is desired, even though communication software is large and complex.

All users have been provided with the same communication services up to now. From now on, however, users expect to be provided with individual services. Today, communication networks are evolving toward open networks where the users themselves develop services.

This situation requires a technique enabling users who do not have detailed knowledge about communication networks or communication software, to develop new communication services. The proposed method of automating software generation is characterized as follows.

- To generate software by describing specifications that can be grasped by communication service users.
- To allow the addition of new services even without knowing the details of existing software.
- To generate error-free, reliable software.

To achieve our purpose we present a communication software generation method that derives satisfactory service specifications from initial requirements described by users, and generates software that satisfies the service specifications. We focus on the part of communication software that directly participates in providing users with services. In other words, the target of our automatic generation is software that needs to be newly developed to handle service additions. We do not consider operating systems nor maintenance and administration software.

The proposed method of automating communication software generation is illustrated in Fig. 1.1. Service designers describe requirement specifications by a specification language called STR (State Transition Rule) [1]. The described requirements may be incomplete and may sometimes have contradictions. The requirement specifications are transformed into "complete" specifications that satisfy the requisites for communication services. Service designers may interact throughout the transformation. This transformation is called specification completion.

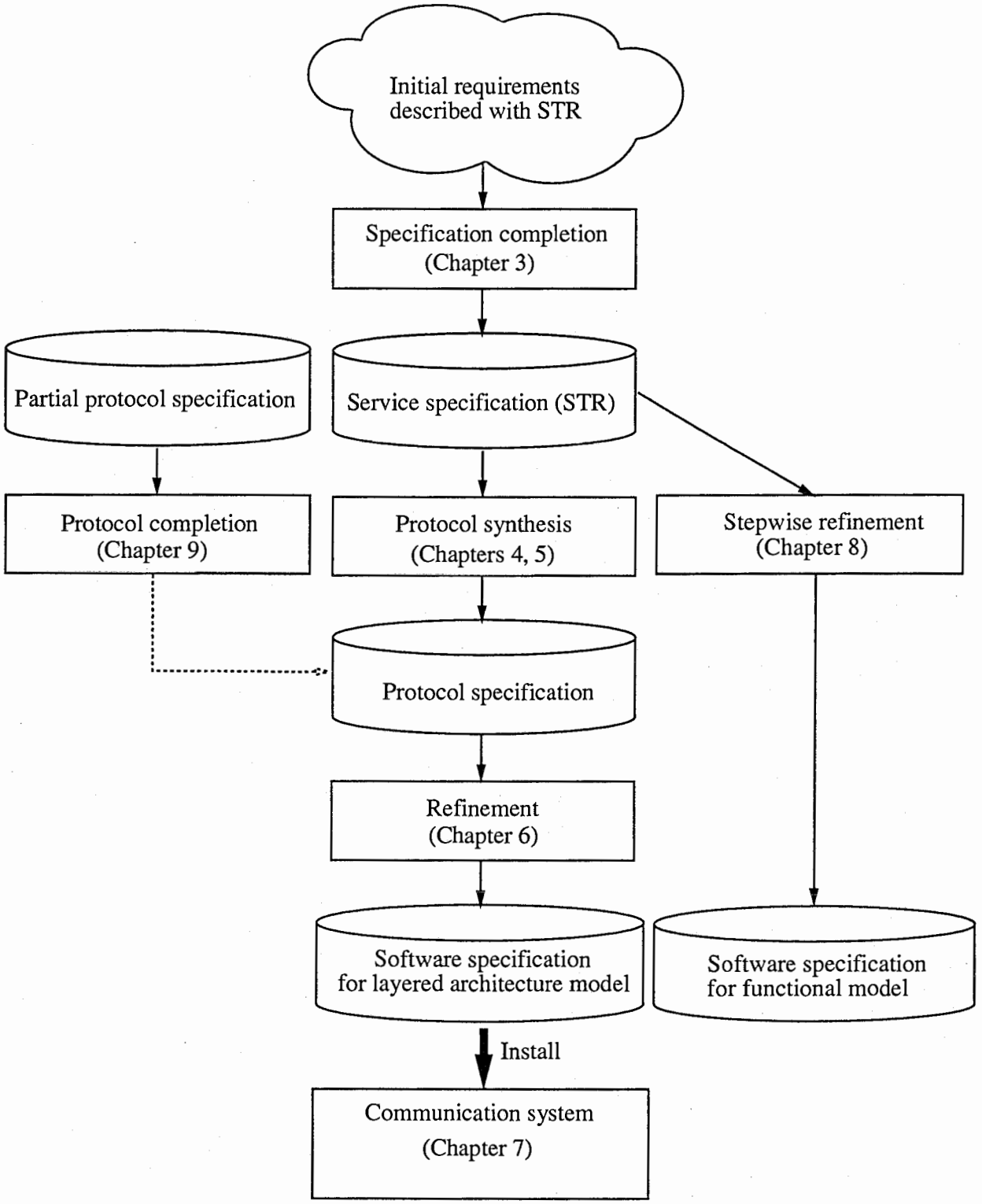


Figure 1.1: Flow of Software Generation.

The usual development style for communication software is adopted to add a new service to existing services. Consequently, it is crucial to detect and resolve all conflicts among these services [2]. Specification verification methods [3][4][5] have already been proposed for specifications described by STR; specification verification is out of the scope of this thesis.

The obtained service specifications are synthesized to produce protocol specifications. In this synthesis a layered architecture model is adopted as the protocol architecture. The OSI reference model [6] is typical of the layered architecture model. Communication services are provided to users through switching systems and telephone terminals. The synthesized protocols only describe the rules of message exchanges between protocol entities; they do not include the control specifications driving such communication equipment. Then, the protocol specifications are transformed into software specifications furnished with the control specifications of the communication equipment. After the obtained software specifications are coded and then installed on the communication system specified by the layered architecture model, the system can provide users with communication services.

A functional model [7] has been standardized as an architecture for providing communication services. It is a distributed architecture in which each function constitutes a protocol entity. Software specifications conforming to the architecture are obtained by stepwise refinement.

The above software generation methods are oriented for users who are non-experts in communication systems or communication software. A field, however, does exist for experts to generate software by implementing given protocol specifications. Message sequence charts [8] are proposed for describing these protocol specifications; such charts describe specifications that can automatically be synthesized to produce protocol specifications [9].

However, synthesized protocols may include exceptional behaviors which do not correspond to any of the requirement specifications. This in turn may cause protocol errors such as deadlock states. The resolution of such protocol errors from exceptional behaviors is called protocol completion.

This paper proposes methods to automate these subjects. There is no system to automate all of them consistently. In other words, this is the first research on generating communication software from service specifications, which can be described without detailed knowledge, through protocol synthesis, after which the generated software is actually used for implementing communication services. There are two types of specification description methods. The internal behavior of a communication system is usually described in the form of specifications to produce communication software. In protocol synthesis methods, service specifications are described by regarding a communication system as a black box. These methods have not been applied to transform service specifications into executable communication software. On the other hand, in this research, incomplete service specifications are transformed into implementable specifications as communication services, and finally to communication software.

The proposed methods are incorporated into the software generation system in Fig. 1.2, and characterized as follows.

1. Service specifications are described as a set of state transition rules for terminals observable from outside of a communication system.

2. Errors in each rule and insufficient rules are detected, and corrected or complemented.

3. The method can be combined with validation and verification methods for specifications.

4. Protocol specifications are synthesized from service specifications.

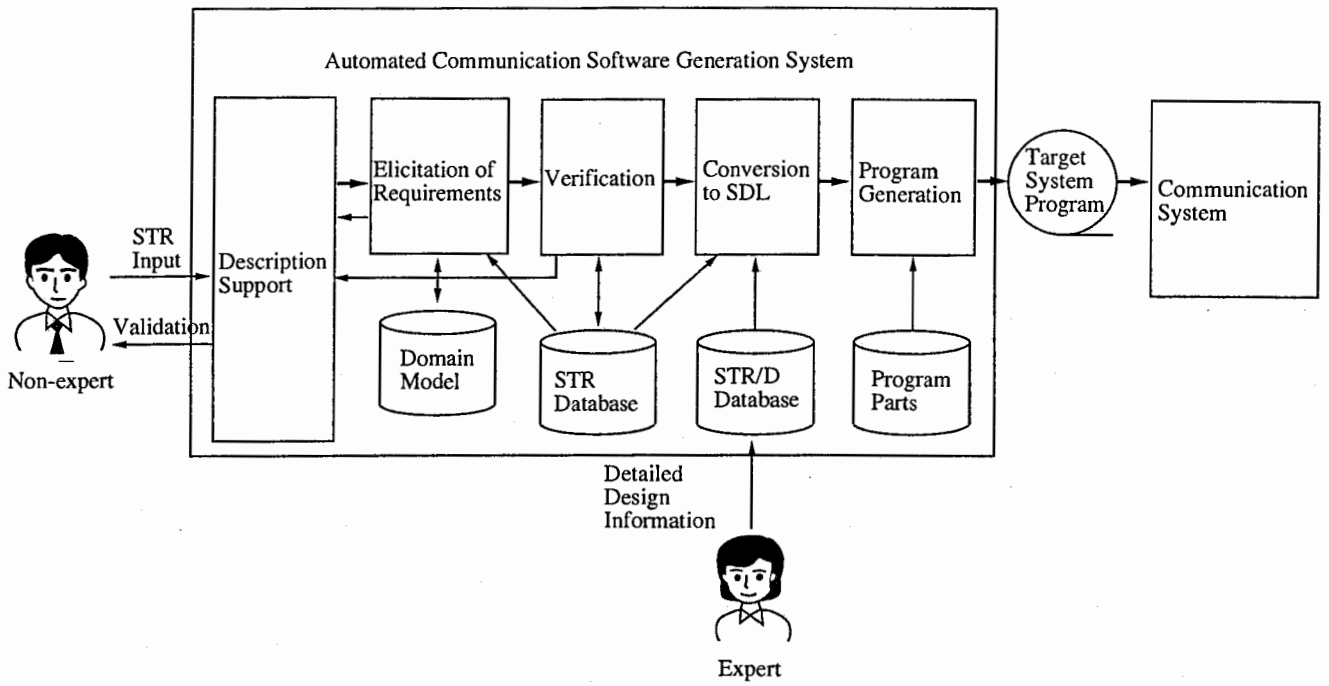


Figure 1.2: Communication software generation system.

5. Detailed specifications are described as knowledge.

The following effects are expected from communication software generation with the above features.

1. Communication software can be obtained starting from incomplete service specifications. This implies that communication service specifications can be obtained from ambiguous and fragmentary requirement specifications which are imagined by the users themselves.
2. Communication software reflects the user's intention. Therefore, the behavior of the generated software corresponds to the user's intention.
3. Specification errors can be detected.
4. Communication service specifications are described by non-experts. Non-experts are defined as people who do not have knowledge of the communication software architecture, protocols and control methods of communication systems.
5. Experts scarcely participate in automated software generation from users' requirement specifications.
6. The size of the specifications is small. This implies that services are expected to be developed in a short time.

Concerning the method for making feature 1 come true, its details and limits are clarified in Chapter 3. Feature 2 is achieved by itself because the specifications that are validated by users are automatically transformed into communication software. Feature 3 is achieved both

by verifying a requirement specification and by detecting and resolving conflicts between the requirement specification and existing specifications. We note that the detection and resolution of interference between specifications are out of the scope of this thesis. The proposed method is shown to have features 4, 5 and 6 by the results of experiments in Chapter 7.

Specification languages are indispensable for automating the generation of communication software. For this purpose, a specification language called STR is used in this thesis. The reason for using STR is described in Section 1.2, which also gives an informal description of STR. Sections 1.3 to 1.7 below describe problems to be solved and related works. Section 1.8 gives an overview of this thesis.

1.2 Requisites for a Specification Language

A specification language is necessary for describing requirements and automating software generation. We clarify requisites for a specification language that achieves these purposes.

- Service specifications independent of implementation.

  Service specifications should not include specifications of the architecture nor functions provided by the target system. When one uses a communication service, any behavior performed in the environment surrounding the communication system turns into a specification of the communication service.

- New service addition without detailed knowledge of existing specifications.

  New communication services are usually added to existing services. Therefore, it is desirable to be able to describe specifications of new services without knowing existing services.

- Automatic detection of conflicts.

  Contradictions are sometimes included in specifications and there may be conflicts between new specifications and existing specifications. These undesirable phenomena can be detected in the specification description phase.

- Stepwise refinement from specifications to programs.

  Specifications are gradually refined into software.

Many specification languages including SDL [10] and LOTOS [11] have been proposed for specifying communication software or protocols [12][13][14]. SDL is the most commonly used language for the formal specification of telecommunication system behaviors. Much of the recent work in this field has centered on LOTOS. These languages can be used to describe implementation independent specifications at an appropriate level of abstraction. They incorporate an explicit notion of execution order or synchronization among processes. This implies that the composition of specifications needs them to be adjusted. On the other hand, in some rule-based languages specifications are simply composed by the logical conjunction of rules. In other words, specifications can be augmented simply by adding new rules.

Rule-based specification languages L.0 [15], FRORL [16] and STR [1] were proposed in the field of communication service specification. Cameron et al. [15] used rule-based language L.0 to implement a real-life protocol. Tsai et al. [16] used frame-and-rule oriented requirement

dial-tone(A), idle(B), m-cfv(B,C), idle(C)
dial(A,B):
ring-back(A,C), ringing(C,A),
ping-ring(B,A), m-cfv(B,C).

Figure 1.3: An example STR rule.

```
R1) idle(A) offhook(A): dial-tone(A).
R2) dial-tone(A),idle(B) dial(A,B): ring-back(A,B),ringing(B,A).
R3) dial-tone(A) wrong-dial(A): busy(A).
R4) dial-tone(A),not[idle(B)] dial(A,B): busy(A).
R5) ring-back(A,B),ringing(B,A) offhook(B): path(A,B),path(B,A).
R6) path(A,B),path(B,A) onhook(A): idle(A),busy(B).
R7) busy(A) onhook(A): idle(A).
R8) dial-tone(A) onhook(A): idle(A).
R9) ring-back(A,B),ringing(B,A) onhook(A): idle(A),idle(B).
```

Figure 1.4: STR description of pots.

specification language FRORL. Hirakawa and Takenaka [1] proposed STR as a specification language for communication services. In the methods of L.0 and FRORL, specifications are incrementally refined to obtain protocol specifications; however, they cannot synthesize protocol specifications from service specifications. A major difference between STR and these two languages is that STR supports conflict detection and protocol synthesis as proposed in this thesis.

Communication service specifications can be described by specifying terminal behaviors which can be recognized from outside the communication system(s). With STR, we can describe specifications without detailed knowledge of the target system or implementation dependent factors. This means that STR is one of such languages able to satisfy the above requisites. In this thesis STR is adopted as a specification language.

We give an informal explanation of STR; a precise definition is given in Section 2.3. Figure 1.3 shows an example of an STR rule. The "ring-back(A, C)" represents that terminal A has the relation "ring-back" to terminal C. The "dial(A, B)" is an event at terminal A. This rule shows that if a user on dial-tone receiving terminal A dials terminal B, which implements the call forwarding service to terminal C ("m-cfv(B, C)"), and terminal C is idle, then the call to terminal B is forwarded to terminal C and the states of terminals A, B and C are changed to the ring-back tone receiving state (A), ping-ring receiving state (B), and ringing state (C), respectively.

We next describe an example of a service specification (Fig. 1.4). This service specifies a basic telephone service between two telephones, called the plain old telephone service (pots).

The prefix of each rule is the name of the rule. We briefly explain the specification.

R1 When a telephone is picked up, its state changes from the idle state to the dial-tone receiving state.

R2 When a user makes a call to an idle telephone, the state of the calling telephone changes to the ring-back receiving state and the state of the called telephone changes to the ringing state.

R3 If a user dials a wrong number, the state changes to the busy tone receiving state.

R4 If a called telephone is being used (i.e., not idle), the call cannot make a connection and the state changes to the busy tone receiving state.

R5 If the called party answers the phone, the call changes to the talking state.

R6 If one of the talking parties hangs up, the state of its telephone changes to idle and the other telephone changes to the busy tone receiving state.

R7 If a telephone in the busy tone receiving state is hung up, the state returns to the idle state.

R8 If a user hangs up before dialing, the telephone returns to the idle state.

R9 If the calling party hangs up while the called telephone is ringing, the states of both terminals change to the idle state.

Each rule describes a terminal behavior that is observable from outside a communication system. In this thesis we use STR as a service specification description language; however, the results are not inherent to STR. The results can be widely applied for specification languages described by production rules.

1.3 Specification Completion

Errors at an early stage of software development cost more to debug than those at a later stage [17]. Requirements acquisition is the most upstream development process. Nevertheless, the system support for requirements acquisition is delayed compared with other development phases'.

Users do not always have precise requirements. It is therefore inevitable that user requirements contain ambiguities, insufficiencies and even contradictions [18]. Considering this, it is indispensable to support a specification completion method that derives service specifications from such problem requirements. One of the objectives of this research is to obtain consistent and complete specifications from such problem requirements.

A lot of different research on specification languages start on the premise that user requirements are defined definitely as computer-processable formal specifications. In other words, research on formal languages has been focusing on automation, to design software satisfying user requirements. Specification verification is one of the support items being studied. Specifications have to be verified that they satisfy constraints arising from the target system.

Research concerning automation techniques starting from formal languages has been called design engineering. On the other hand, requirements engineering starts on the premise that it is difficult to elicit specifications from users. Requirements engineering consists of requirements acquisition from users and specification validation. Acquired specifications are validated by users.

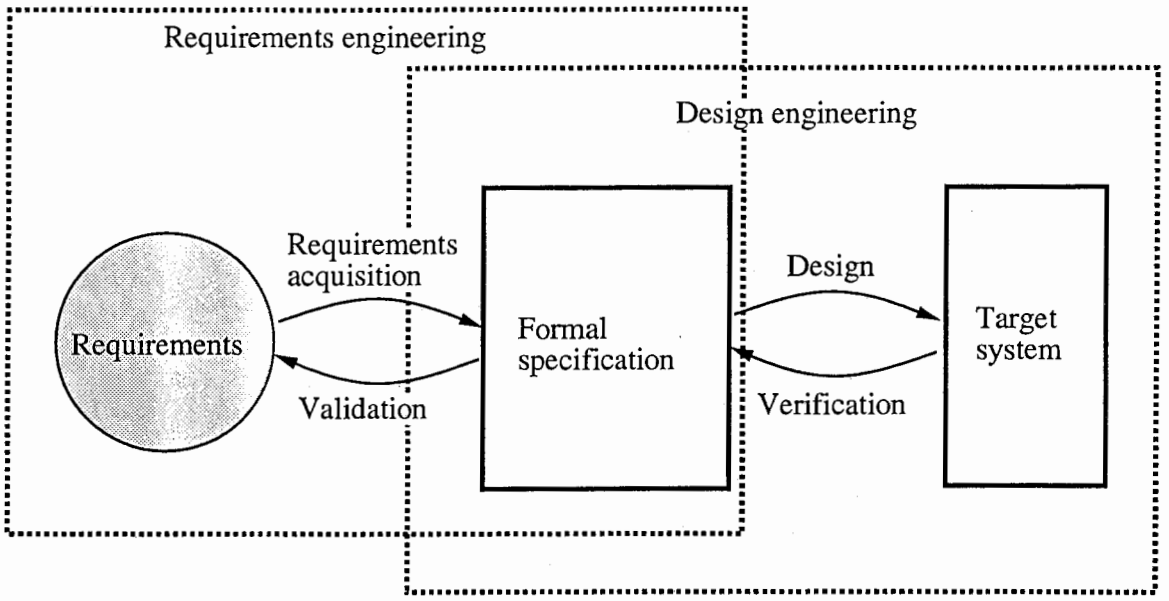


Figure 1.5: Requirements engineering and design engineering.

Figure 1.5 illustrates the relation between requirements engineering and design engineering from [19][20]. They are characterized as follows. Requirement specifications obtained by requirements engineering describe behaviors of target software systems in their surrounding environments. System specifications resulting from design engineering describe functions and the architecture of the target system.

It is crucial to eliminate errors in the requirements acquisition phase to reduce software development costs. Requirements acquisition is therefore divided into two phases: elicitation and formalization. In the elicitation phase user requirements are clarified and represented as specifications. Requirements elicitation is a brain-work session of requirements acquisition. It is almost impossible to automate requirements elicitation. Support for requirements acquisition differs greatly when formal languages are used and when they are not.

Requirements Apprentice [18] and the software design support system based on FRORL are assumed to separate end-users and requirements analysts who describe requirements by a formal language. In our approach, our aim is to enable users or non-specialists of communication systems to describe their own services. Therefore, we intend to present a requirements acquisition method assuming that requirement specifications are to be described by users rather than requirements analysts.

Requirement specifications sometimes contain ambiguities and errors, and sometimes rules are missing. One of our objectives is to establish a support method obtaining well-formed communication service specifications from requirement specifications. This support consists of two phases. One phase is to detect rule errors and missing rules. The other phase is to modify and supplement specifications for transforming original specifications into satisfiable specifications.

Domain knowledge plays a key role in supplementing service specifications. However, using domain knowledge in specification completion poses problems: how to use what kind of knowledge, and how to express and how to acquire new knowledge. Domain knowledge usually includes common knowledge about the environment surrounding a software system, knowledge

on specifications described in the past, and knowledge on analysis and design methods [19]. In this thesis an abstraction of existing communication services is used as domain knowledge. We do not have to use the same knowledge on analysis and design methods as that used in ASPIS [21], because the domain knowledge here is used only for the purpose of supplementing insufficient specifications. We assume that the domain knowledge is supplied by experts of communication services. For our purpose domain knowledge should be well supplied. Therefore, the method used in BLIP [22] to augment domain knowledge starting from sloppy modeling cannot be used. These works can be characterized as application-independent. Application-independent environments for specifying arbitrary software systems, however, are still years away.

WATSON [23] operates in finite-state reactive systems: those whose most important behavior requirement is the association of particular sequences of input stimuli with corresponding sequences of output responses. A communication system is one of such examples. The domain knowledge used in WATSON includes knowledge about telephone hardware, network protocols, expected end user etiquette, exception handling, time-outs, and preferred styles of control skeleton design.

In this research domain knowledge is used for supplementing rules lacking in the users' initial requirements. The main difference between the approach of WATSON and ours has to do with the abstraction level of domain knowledge. In our approach knowledge on communication services is abstracted in the domain knowledge rather than terminal behaviors. Therefore, new services can be supported when specifications are described.

1.4 Protocol Synthesis

A communication service provides information exchange between multiple users. For smooth exchange rules are needed to govern the interactions between communication entities. Such rules are called protocols. In communication software a protocol determines the outline of a control flow. Then, protocol synthesis from a service specification is performed as one step of the automatic software generation.

Protocol architectures have to be defined when synthesizing protocol specifications. There are standardized protocol architectures such as the OSI reference model, signaling system No. 7, etc. In most of them the communication functions are layered. Each layer consists of a collection of protocol entities (or protocol processes) that are distributed over different locations. Figure 1.6 illustrates a layered protocol architecture model. Protocol entities in the same layer are called peer entities or communicating entities. The peer entities of layer N provide the communication services called N-services to layer N+1 users. The services provided by layer N are accessed by user entities through a layer interface called service access points. Likewise, user entities of layer N access the communication services, called (N-1) services, provided by the layer below through another layer interface. The entities of layer N use these services for exchanging messages. The rules that govern the exchange of these messages among the entities are collectively called an N-protocol.

We show related works on protocol synthesis assuming a layered architecture model. Probert and Saleh [24], and Ichikawa and Takami [25] have surveyed protocol synthesis methods. Two kinds of protocol synthesis methods are known for initial specifications. One type starts protocol synthesis from partially-defined protocol specifications and the other type starts from complete service specifications.

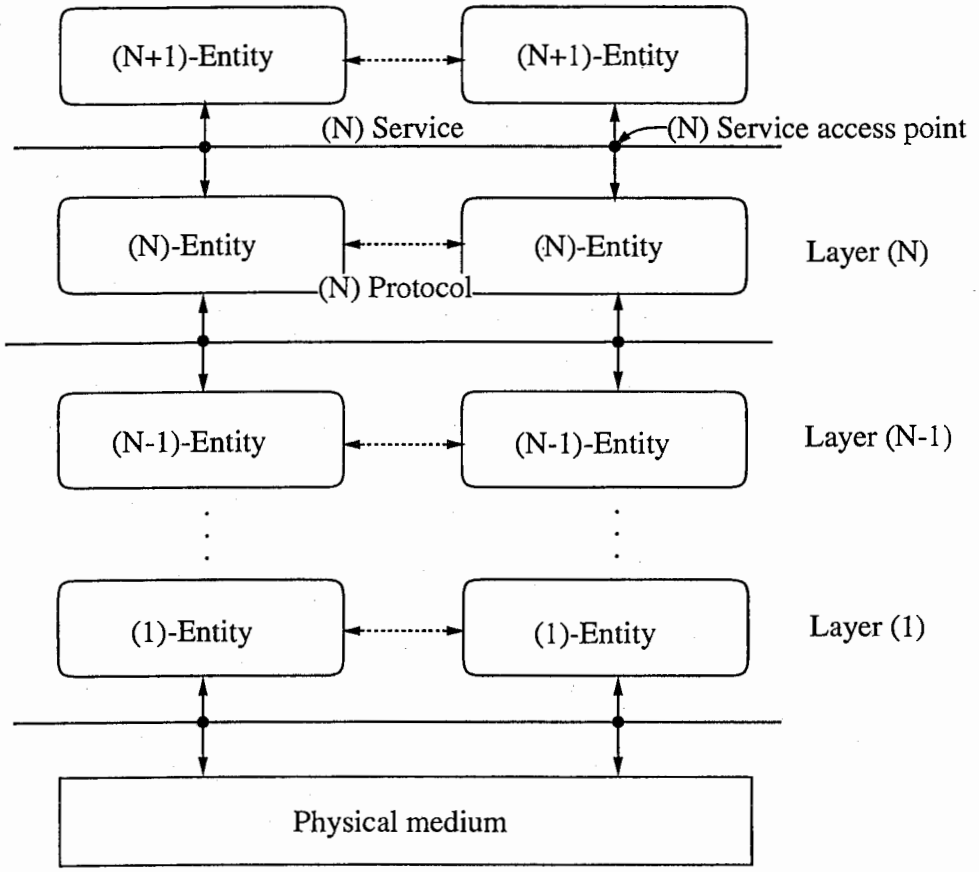


Figure 1.6: Layered protocol architecture.

The former methods are classified into three types. Merlin and Bochmann [26] proposed the submodule construction method. In this method, if a system is to consist of n submodules and the system as well as (n-1) submodules are specified, then the method determines the specifications of the additional nth submodule. A lot of research [27][28][29][30][31] has been done to synthesize error-free protocol specifications starting from erroneous and incomplete protocol specifications represented by Finite State Machines (FSM). Other approaches [9][32] synthesize error-free protocol specifications from multiple partial protocol specifications described by message sequence charts. These three kinds of methods basically synthesize protocols by inserting missing primitives such as message send and receive or coordinating communications.

The latter methods are classified into two types concerning specification description. Bochmann and Gotzhein [33][34] describe service specifications by a LOTOS-based language. Chu and Liu [35], Saleh and Probert [36], and Kakuda et al. [37] adopt FSM-based description. These methods synthesize protocols by inserting appropriate interactions among processes to provide given service specifications.

The protocol synthesis methods described in this thesis use FSM-based description for service specifications. Concerning protocol synthesis methods from service specifications, we clarify the major difference between the described methods and methods proposed in literature. Note that conventional methods describe the execution order of events at service access points that are previously defined. In our methods, on the other hand, service specifications are described with the rule-based and FSM-based language STR. With this description method two features are used to express the wide-range of communication services. A service access point is expressed by a variable representing a terminal. If there is a partial state satisfying a rule application condition of a rule, the rule is applied to the partial state in the whole system state. Because of the features, it is not necessary for the number of terminals participating to be bound as long as the number of terminals appearing in the rule application condition is finite. This implies that we can describe a communication service that permits an indefinite number of participants such as a conference service with unlimited participants.

Furthermore, in telecommunication systems many events can occur asynchronously at different service access points. In this case, our methods synthesize protocols implementing distributed algorithms [38] that satisfy given service specifications. In other words, when an event occurs at an SAP, its protocol entity has to communicate with other processes by necessity to know their states. No such protocol synthesis has been previously proposed in literature for a layered architecture model.

When a user operation at a terminal, called an event, occurs, a synthesized protocol determines an appropriate rule by exchanging messages among protocol entities. A condition of a rule application is represented as a rooted labeled directed graph. The state of the communication system itself is represented as a directed graph. Consequently, a rule is determined by investigating if its application condition graph is included in the communication system graph at that time.

We propose two examination methods. One method examines rule application conditions sequentially and the other one examines them in parallel. We can synthesize protocol specifications that perform these two methods.

The synthesized protocol implements a distributed algorithm to find a subgraph isomorphic to a rule application condition graph in the whole graph. Such a subgraph isomorphism problem is one of the NP-complete problems [39]. In an ordinary communication service, the scope to be searched is usually not large. Therefore, the synthesized protocol works in a practical time. Actually, there is no problem in practical use [40].

1.5 Refinement from Protocols to Software Specifications

The above protocol synthesis methods suit the layered architecture model. This kind of protocol synthesis is available for software generation when there is no constraint on functional distribution like with PBX systems. We show that protocol synthesis based on a layered architecture is useful for generating PBX software by implementing several communication services on a PBX.

Protocol specifications describe message exchange rules between protocol entities. They do not involve detailed specifications such as control specifications for actual communication systems. The synthesized protocol specification needs to be refined in order to generate executable software on a communication system. This refinement has been performed manually [15][16][41]; and it has needed detailed knowledge about communication systems and communication software.

In this research we aim at generating communication software without using such detailed knowledge. Therefore, we define a detailed specification language STR/D (Detailed Specification Language for STR) [42] so that detailed specifications can be described as knowledge independent of service specifications. This means that detailed specifications are described as knowledge beforehand. Note that if such predescribed knowledge is insufficient in refining protocol specifications, then an expert on communication systems helps by adding detailed knowledge. A new method [43] is proposed to generate such knowledge.

The communication software generation method described above has been shown to be feasible in an application to develop actual PBX software. Several typical communication services have been implemented by the automatically generated software.

Communication systems such as PBX systems have different control interfaces depending on their vendors or machine types. While generating different software for each communication system, we defined a logical interface common to the PBX systems and generated communication software conforming to the interface. We provided interface conversion programs to fill the gap between the PBX dependent control interfaces and the logical interface.

There is an interface called CTRON (Central and Communication The Realtime Operating System Nucleus) for running the same application program on heterogeneous communication systems rather than for automating software generation. If such an interface were installed on a PBX, we might be able to adopt the interface as the logical interface.

In this research we defined a logical interface that can be used to define meanings of state primitives on communication systems. Using this interface we have implemented several communication services on two PBX systems. We show an implemented software architecture and the defined logical interface, and also show results giving the efficiency of generated software.

1.6 Stepwise Refinement for Functional Models

There is another protocol architecture called the functional model [7]. Universal Personal Telecommunication [44] has been standardized to be provided using the functional model in Fig. 1.7. In the functional model functions are not layered but distributed. These functions are distributed in functional entities.

Conventional protocol synthesis methods including the above described protocol synthesis methods are based on the layered architecture model in Fig. 1.6. In this thesis we propose a

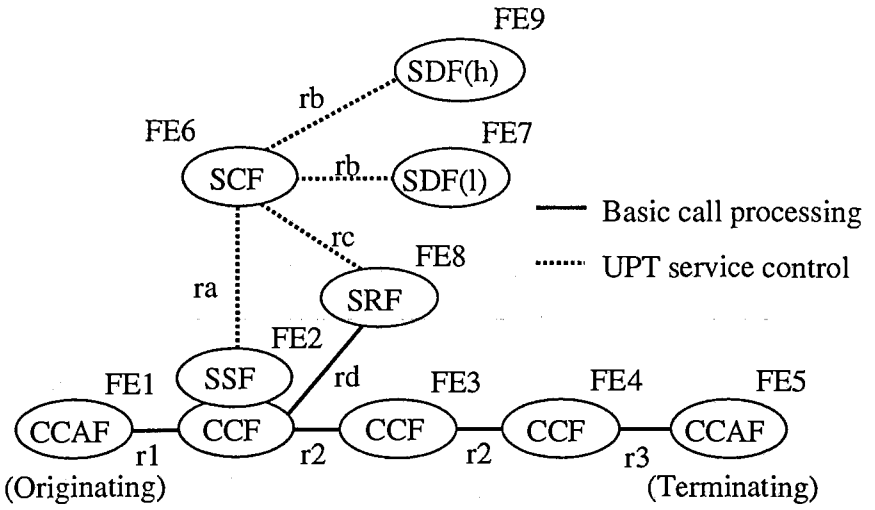


Figure 1.7: Functional model for UPT service set 1 provision.

software generation method that is adapted for an architecture with layering and functional distribution. The method is a semi-automatic refinement method, however, a new more automated method [45] has been proposed for this refinement problem.

1.7 Completing Protocols

Protocol synthesis is a powerful technique to automate communication software generation. In this approach a layered architecture model is adopted as the protocol architecture. The OSI reference model [6] is typical of this layered architecture model. Service specifications are defined as distributed functions provided by a communication system to its users. A communication system can be viewed as a service provider which offers some specified communication services to service users who access the system through geographically distributed service access points. Protocol synthesis is a very promising way of developing a reliable protocol. However, protocol synthesis cannot be applied to the case where the protocol itself is specified even in the context of the layered architecture model. In other words, protocol synthesis is likely to produce protocols that are different from those that the protocol designer wishes. This problem is solved by describing protocols themselves as service specifications, or requirement specifications.

Message sequence charts [8] are standardized to specify protocols. For the above purpose, the service specifications are defined as sets of partial specifications of protocol specifications described by message sequence charts. Service specifications described by message sequence charts can be synthesized to protocol specifications [9], where a protocol is represented by a set of communicating finite state machines with FIFO (First-In-First-Out) channels [46][47]. This protocol synthesis method guarantees the following three properties: requirement specifications are feasible, required behaviors are included in a synthesized protocol, and there are no nondeterministic behaviors. A synthesized protocol may include exceptional behaviors not corresponding to any behaviors in the given service specifications. Even if the service specifications themselves are verified, these exceptional behaviors may involve protocol errors, such as deadlock states or unspecified reception. Unspecified reception means that there is no action

to receive specific messages being transmitted.

We show a method to transform a synthesized protocol into a protocol that returns to a normal state when it falls into an exceptional behavior. Protocol completion refers to the case where a synthesized protocol has been transformed into an error-free one. We can obtain an error-free protocol from error-free service specifications by synthesizing and then completing the synthesized protocol.

Zafiropulo et al. [27] proposed a protocol completion method from given protocol specifications by letting send and receive events correspond to one another. Kakuda [48] proposed a method to add rollback sequences to checkpoints. These methods do not explicitly use the concept of service specifications, or requirement specifications. Consequently, protocol completion finishes when a protocol is transformed so that every possible behavior reaches the final state with empty channels in due course.

In this thesis, we clarify the scope within which a protocol can be completed without modifying service specifications, and propose a protocol completion method that modifies the smallest number of service specifications. If the tactics to tackle exceptional behaviors are given by rules, the method automatically completes protocols. We show the effectiveness of the method by synthesizing the X.227 protocol from its partial specifications. Since in our method service specifications are explicitly used, protocol completion can also be applied to resolve undesirable feature interactions [2] appearing as exceptional behaviors.

1.8 Overview of the Thesis

We give an overview of this thesis.

In Chapter 2 we give preliminary definitions. We introduce the layered architecture and the functional model as network architectures that are used when synthesizing protocols from service specifications. Service and protocol specifications are defined on the network architectures. Then we introduce a rule-based specification description language STR. In STR services are specified as state transition rules of terminals connected to a communication system. A rule specifies state transitions of associated terminals when a user operates a terminal.

In Chapter 3 we discuss a specification completion method[49][50][51] concerning communication service specifications described by rules. First we define a service formally and show errors that may be contained in specifications. Reachability analysis is used for detecting such errors. Hypothesis-based reasoning is used for interpolating incomplete specifications. In this interpolation we try to supplement specifications with rules in existing service specifications at first.

If this supplement fails, a communication service model is used for presenting supplementary specifications. The communication service model is obtained as an abstraction of communication services. These communication services have a feature that the number of terminals concerned with one service changes dynamically during operation of the service. Note that there is a problem with how to express a service on the communication service model. In this thesis we represent a service by compositions of fundamental service elements.

Chapter 4 shows a protocol synthesis method [52] suited for a layered architecture. Protocol synthesis is defined as a labeled directed graph rewriting problem. An STR rule describes a graph rewriting rule. The state of the communication system itself is represented as a directed graph. Rule application is achieved by searching and replacing a subgraph isomorphic to a graph representing rule application condition. The subgraph isomorphism problem is one of

the NP-complete problems. However, since a number of terminals do not participate in one communication service simultaneously, combinatorial explosion of the graph searching time does not become a problem in practical use.

A rule is represented as a pair of labeled directed graphs. The state of all terminals in a communication system is also represented as a labeled directed graph. In this graph representation, a communication service specification describes a set of rules to find the appropriate graph representing the condition part of a rule; its resultant is substituted for an isomorphic subgraph in a graph representing the communication system.

In the layered architecture model a protocol entity is assigned to each terminal. Each protocol entity possesses the state of its assigned terminal. When an event occurs at a terminal, its protocol entity initiates communications to determine a rule to be applied. In this chapter we show a protocol synthesis method that generates a protocol entity specification that communicates sequentially among protocol entities.

A graph representing an application condition usually becomes a complex graph with branches. We show a protocol synthesis method [53] that produces a protocol entity specification that communicates in parallel among protocol entities in Chapter 5.

Protocol specifications synthesized in Chapters 4 and 5 are insufficient for implementing communication services on an actual communication system. Protocol synthesis is one step for communication software generation. Synthesized protocol specifications are refined with detailed knowledge about communication software. In Chapter 6 we define a detailed specification description language STR/D [42], and show an example for implementing several services on a PBX.

Chapter 7 shows a result of implementing services on PBX systems [40][54]. The result implies that the proposed software generation method is effective for real-life service development. We can generate communication software independent of target systems by providing logical interfaces for controlling PBXs.

In Chapter 8 we show a communication software generation method [55][56] conforming to a functional model. In the functional model multiple protocol entities are placed at one service access point. We call a protocol entity in the functional model a functional entity. First a service specification is transformed into a set of service specifications of functional entities. From the obtained service specifications we can synthesize protocol specifications of functional entities. This means that a protocol synthesis problem of a functional model is transformed into that of a layered architecture model.

In Chapter 9 we introduce a protocol completion method of protocol specifications synthesized from message sequence charts. This method is available for developing protocol software when a protocol specification is given beforehand. A synthesized protocol may include behaviors that do not correspond to any of the requirement specifications. Such behaviors are called exceptional behaviors. They do not appear until the protocol is synthesized. Since exceptional behaviors can not be verified in the specification description phase, the protocol may fall into an abnormal state such as a deadlock state. We show a protocol modification method that prevents a protocol from falling into an abnormal state [30].

Finally we conclude this thesis in Chapter 10.

Chapter 2

Preliminaries

We introduce the fundamental concepts in this thesis and the specification description language STR. A user is provided with a communication service through a network. The automatic generation of communication software differs depending on what kind of network architecture is used to provide a communication service.

Generated communication software from service specifications has to conform to a target network architecture. To begin with we introduce the type of network architecture. The communication software obeys a protocol defined on the network. The most popular and widely accepted protocol architecture is a layered architecture including the OSI reference model. Our layered architecture is based on two concepts of layering and abstraction. A protocol layer provides the upper protocol layer with services by using services provided by the lower layer.

However, telecommunication services such as Universal Personal Telecommunication (UPT) and Intelligent Network (IN) services are standardized to provide services on the functional model. These two types of architectures are adopted as protocol architectures for protocol synthesis.

Next, we define service and protocol specifications. Communication software obeys a protocol when communicating among protocol entities to provide services. Service and protocol specifications have the relation of requirement and implementation specifications.

Finally, we give a precise definition of STR. The network architecture is a part of the specifications that show how to implement a communication service. This means that service specifications have to be independent of specifications concerning the architecture. Consequently, we describe service specifications independent of architectures and protocols. In this thesis service specifications are described as terminal behaviors observable from outside a communication system. Using this description method, service specifications can be described by non-experts who do not have detailed knowledge of communication systems and communication software. STR is one of the specification languages.

2.1 Network Architectures

We introduce the layered architecture and the functional model. The layered architecture model is illustrated in Fig. 2.1.

In the layered architecture model, a user is provided with services through a service access point (SAP). A protocol entity is assigned to one SAP. A request from a user is received by a protocol entity through a service access point. Upon receiving a request, a protocol entity communicates with other protocol entities to provide the user with a service. Terminals

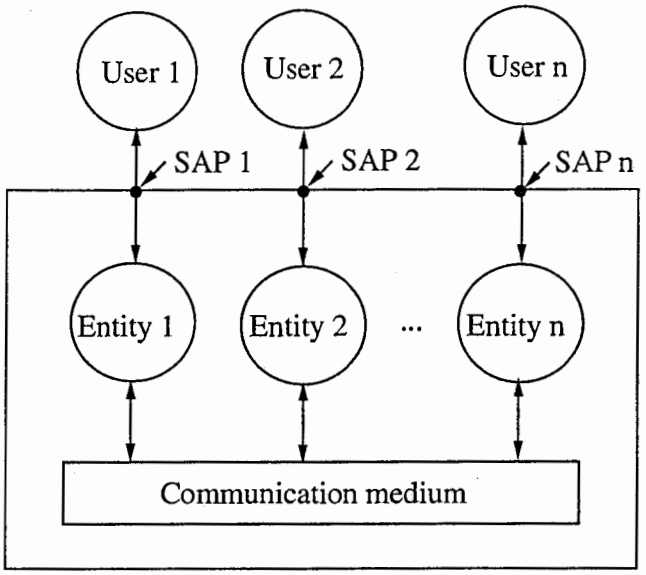


Figure 2.1: Architecture model for layered protocol design.

including telephones should have SAPs in telecommunication services. Communication among protocol entities is assumed to be error-free through a reliable communication medium.

The functional model is a standardized network architecture for providing telecommunication services. In the layered architecture model functions are abstracted by layering. On the other hand, functions are collected into functional entities distributed in a network. Figure 2.2 shows the functional model. In Fig. 2.2 a terminal is connected to CCAF physically. If a service is recognized as UPT, the control of the service is transferred to CCF. Furthermore, some of the stimuli at terminals are received at SRF and SRF responds to them. In comparison with the layered architecture model, the functional model is characterized by the relationship between SAPs and protocol entities as follows:

- Multiple protocol entities correspond to one SAP.
- There are protocol entities that are not assigned to any SAP.

There is a point of view that one SAP corresponds logically to one functional entity. We aim at describing specifications without any knowledge of network architecture; however, we cannot discriminate SAPs corresponding to functional entities.

2.2 Service and Protocol Specifications

A communication system provides communication services for service users who access the system through service access points SAP1, $\cdots$, SAPn. In this modeling service specifications and protocol specifications are defined as follows [57]:

- The *service specification* describes what services the protocol entities of the lower protocol layer provide for their users in the upper protocol layer. The services provided by the lower protocol layer are based on a set of service primitives which describes the operations at service access points through which the services are provided.

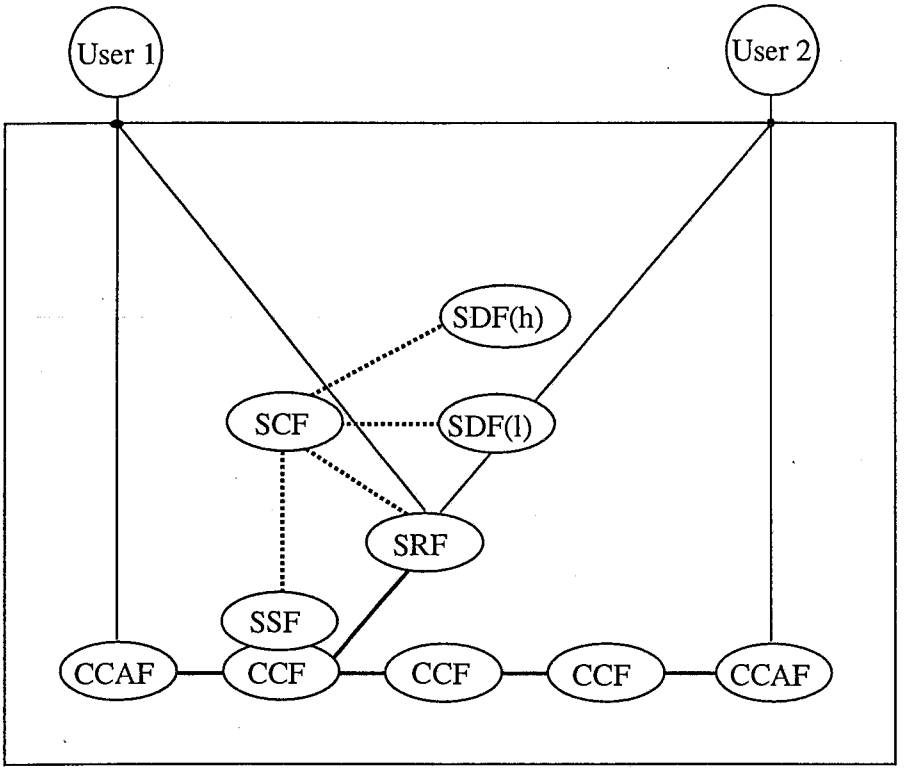


Figure 2.2: Functional model.

- The *protocol specification* describes the interactions among the protocol entities of the lower protocol layer. The interactions are defined in terms of the services provided to the upper protocol layer, and the services available from the communication medium.

In a telecommunication system, users are provided services through terminals, which can be considered SAPs. Within the communication system, protocol entities cooperate to provide services by exchanging messages between entities. This communication between entities is provided by the communication medium. In this architecture, each entity corresponds to just one SAP for a user in the upper layer. This architecture model is an abstraction of the OSI reference model.

On the other hand, IN services are assumed to be provided on the IN CS-1 functional model in Fig. 2.2. In this architecture, there are some entities, e.g., SCF, that do not correspond to any SAP. Since this functional model is an internal architecture, the service designer does not have to take account of the functional model to describe service specifications. This means that service specifications do not and should not have any information about the internal functional model.

2.3 STR

A service specification has to prescribe requirements to be satisfied by a communication service, and should not prescribe other specifications. In communication services, state transitions of terminals for a stimulus satisfy this condition. In STR, state transitions of terminals associated

with a service are described as rules. We give a minimum definition of STR necessary for this thesis.

A service is defined as a set of STR rules. An STR rule consists of three elements: "initial state", "event" and "next state," and has the form:

initial state event: next state.

The "initial state" and the "next state" represent global states of terminals. A global state is represented by a set of local states. A local state is represented by a set of state primitives. A state primitive may have two arguments to express terminal variables. The first argument represents the terminal variable having the state primitive. If the second argument is specified, the terminal designated by the first argument holds a relation of the primitive to the terminal specified by the second argument. Therefore, the local state of a terminal is defined as the set of state primitives whose first argument designates the terminal. A state primitive represents a terminal state which is recognizable from outside a communication system.

The "event" may also have two arguments to express terminals. It represents a logical input to the terminal designated by the first argument. If the second argument is described in an event, this argument represents a terminal identifier given by the event.

A rule may be applied to a set of terminals $t_1, \cdots, t_n$ if its event has occurred at one of the terminals; these terminals have the primitives specified by the initial state of the rule. If there are two rules, `r1`, `r2`, whose state primitives in the initial state are included in the local states of terminals $t_1, \cdots, t_n$, and the initial state of `r1` is included in `r2`, then `r2` is applied. This inclusion relation is not of a total order. Thus, there still exists the possibility that multiple rules may be applied. When multiple rules can be applied, we may select a rule arbitrarily.

In summary, we characterize STR as follows.

- In STR service specifications are represented by terminal behaviors which are observable from the outside. Therefore, we can describe service specifications without detailed knowledge of the communication system inside.
- Since STR is a rule-based language, new services can be added by adding new STR rules, and existing services can be modified by changing the appropriate rules.
- If we define state primitives so that they are independent of specific terminals and networks, we can define communication services independent of terminals.

Chapter 3

Specification Completion of Communication Services

3.1 Problem

Requirement specifications may include errors, ambiguities or be incomplete. It is essential for efficient software development that these problems be removed in the specification acquisition stage. Currently, however, this stage in software development is the least supported among all stages. We show a method of transforming such problem specifications into meaningful specifications as communication services.

In this thesis specification completion is defined as the ability to derive formal communication service specifications from requirement specifications. Of course, requirement specifications are not always sufficient specifications. They may have ambiguities, missing parts or sometimes even be contradictory specifications.

Specification completion of STR descriptions has been considered for detecting insufficient and ambiguous rule descriptions as well as missing rules, and also for solving these errors. Figure 3.1 shows specification completion in STR.

The proposed method uses rules contained in existing services and a communication service model as domain knowledge. This knowledge is assumed to have been provided upon the design of new services. Future work will seek to acquire sufficient service models and to extend them.

To confirm that the completed services satisfy their requirements, service designers must validate the completed service specifications. Animated simulation [58] can be used for this

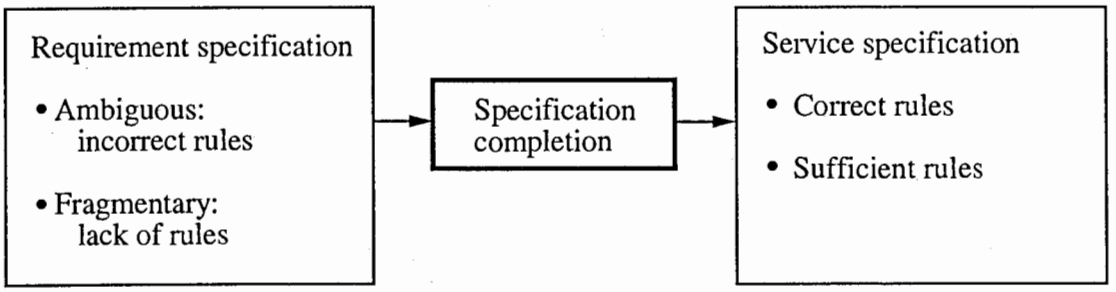


Figure 3.1: Specification completion.

validation.

In communication software development, new services are developed to be added to existing services. In this development style the validity of all service specifications, including existing services, is verified by conflict detection and elimination [3]. Hereafter, it is assumed that specification completion does not need to provide a process for handling errors resulting from the combining of a new service with existing services.

3.2 Services and Requirements

We define a service in STR. Let S be a set of STR rules. If there is a set of terminals T that satisfies the following three conditions, S is called a *service.*

Let α be a global state of T such that every terminal in T is in its initial state, i.e., idle. Let β be a global state of T that is reachable from α by using rules in S. If a set of rules satisfies the following three conditions, S is guaranteed to be syntactically correct.

1. α is reachable from β.
2. There is exactly one applicable rule r at β.
3. For every rule r there is a global state β that is reachable from α and where r can be applied.

The first condition guarantees no deadlock state, no livelock. The second condition guarantees no nondeterministic state. The third condition guarantees no unnecessary rules. Note that S is not always meaningful as a service. The validation that S is a meaningful service is left to a service designer.

3.3 Detection and Elimination of Specification Errors

Errors in requirement specifications are classified into three types: description errors in rules, contradictions among rules and insufficiency of rules. Our purpose is to obtain service specifications from such problem requirements. We show a method that detects these errors and eliminates them. The procedure is illustrated in Fig. 3.2.

Step 1 Detect the errors in each rule and then correct them by consulting a service designer.

Step 2 Detect missing rules. This is done by reachability analysis from an initial state back to the initial state. If a deadlock state is detected, it is determined that some rules are missing.

Step 3 Ask a service designer to input an additional requirement to designate a state to be reached from the detected deadlock state.

Step 4 Supplement rules by utilizing rules used in existing services. This supplementation uses hypothesis-based reasoning.

Step 5 When an insufficient specification is not supplied with existing rules, generate new rules by using a domain model, that is, an abstraction of communication services.

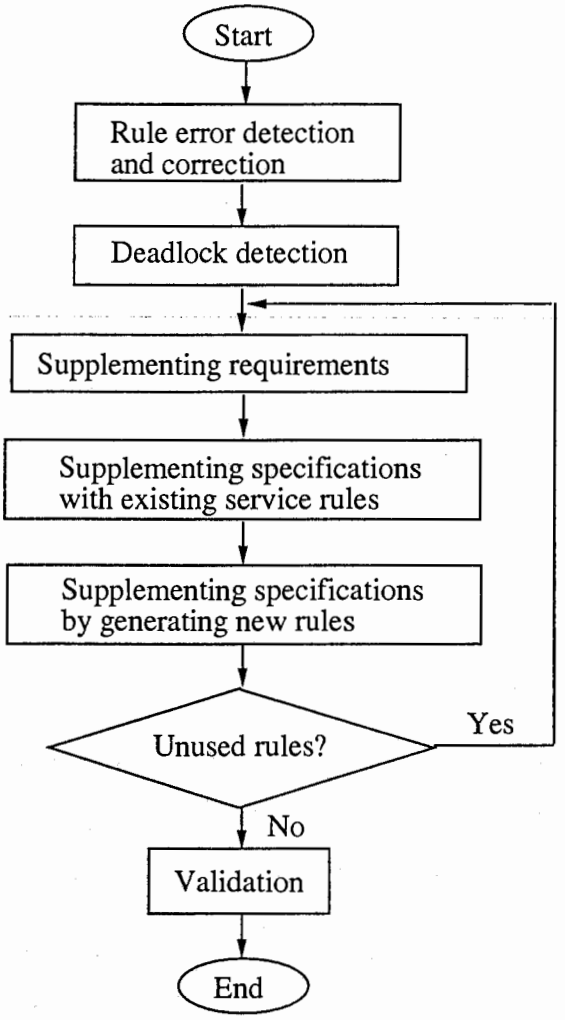


Figure 3.2: Specification completion.

Step 6 Examine if all rules are being used. If unused rules exist, ask a service designer if the unused rules are being used for additional requirements. If the answer is affirmative, go to Step 2; else discard the unused rules.

Step 7 Validate if the completed specifications have been accepted. Animated simulation of specifications is used in this validation. If the specifications have not been accepted, try specification completion again.

Details of steps 2 to 5 are explained in the succeeding sections.

3.4 Detection and Correction of Rule Errors

Rule description errors are classified into two types. One is the case that there are missing primitives or wrong primitives in a rule. The other is the case that a rule itself is correct, but it contradicts with other rules. These errors are detected as nondeterministic variables, free variables or nondeterministic rules [59]. We define these errors as follows.

Nondeterministic variables A rule in which there are multiple possible next states when the rule is applied.

Free variables A rule with a variable that cannot be trodden from its event variable representing a terminal where an event occurs.

Nondeterministic rules Two rules with the same initial state and event, but with different next states.

We exemplify these errors. First, we show an erroneous rule with nondeterministic variables. This rule has symmetrical variables B and C for A. When the rule is applied to an actual state
"path(a,b),path(b,a),path(a,c),path(c,a)"
for terminals a, b and c, there are two possible next states:
"path(a,b),path(b,a),hold(a,c),hold(c,a)" and
"path(a,c),path(c,a),hold(a,b),hold(b,a)".
They are nondeterministic behaviors.

path(A,B),path(B,A),path(A,C),path(C,A)
flash(A):
path(A,B),path(B,A),hold(A,C),hold(C,A)

Next, we show a rule error of free variables. In this rule C of "idle(C)" in the initial state is not identified with the initial state "path(A,B),path(B,A),idle(C)" and the event "flash(A)".

path(A,B),path(B,A),idle(C)
flash(A):
hold(A,B),hold(B,A),ring-back(A,C),ringing(C,A)

Finally, nondeterministic rules are shown. The first rule specifies that dialed terminal B is directly connected from A in the hot-line service. The next rule specifies that dialed terminal B is called.

dial-tone(A),idle(B) dial(A,B): path(A,B),path(B,A)
dial-tone(A),idle(B) dial(A,B): ring-back(A,B),ringing(B,A)

These errors are automatically detected; however, correcting them is left to interaction with a service designer.

There is another kind of errors in which a description is correct as a rule but the rule is not suited to a service element. Errors of this type are detected and corrected at the point of verifying whether requirement specifications satisfy service conditions.

3.5 Detection of Insufficient Rules

We show a method of detecting missing rules as a service specification. If missing rules exist, deadlock states will appear when the requirement rules are simulated starting from the initial states. Detection of these deadlock states is done by a state enumeration method of conflict detection [3].

When deadlock states are detected by simulation, the specification completion system lets a user input additional requirements specifying states to be reached from the detected deadlock

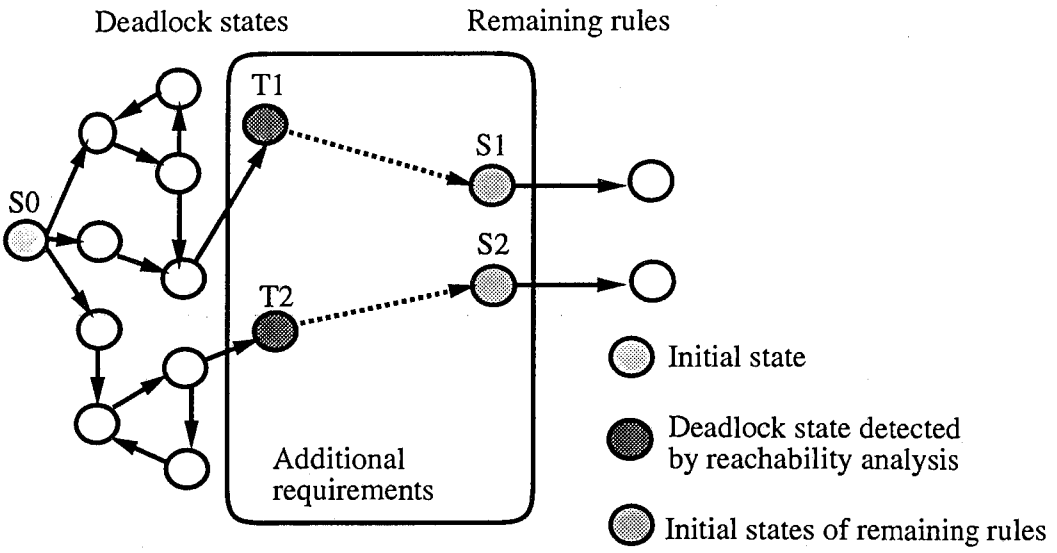


Figure 3.3: Reachability analysis for detecting missing rules.

states. Figure 3.3 shows such a situation. Deadlock states T1 and T2 are detected and then additional requirements are input so that S1 and S2 can be reached from states T1 and T2, respectively. Target states S1 and S2 are specified by the user or selected from the current states of remaining rules that have not yet been used.

When enumerating reachable states within requirement specifications, the number of usable terminals is always determined before starting the enumeration. Therefore, there is a limit: there are undetectable deadlock states that appear only when more terminals, than the predetermined number of terminals, are permitted. However, the necessary number of terminals can be calculated under some constraints [60][61].

3.6 Supplementing Insufficient Rules

If an additional requirement specifies that state B is to be reached from state A, the requirement specification is supplemented with a set of rules $R_1, \cdots, R_i$ that enable the transition from a state X ($\supset$ A) to a state Y ($\supset$ B). Figure 3.4 illustrates the hypothesis-based reasoning when deadlock state A is detected and a new requirement that B is to be reached from A is added. The hypothesis to be added to A has the next two constraints.

- Hypothesis X-A is empty, or a state X-A can be reached from a part of the initial state in Fig. 3.4, say I, independent of the transitions from the remaining part of the initial state, say J, to A.

  This implies that X-A is reachable from I without affecting reachability from J to state X-A. Owing to this constraint, deadlock at A is resolved.

- X-A and Y-B are consistent with A and B, respectively.

  When A (B) involves state primitive p, $\neg$p cannot be hypothesized to X-A (Y-B).

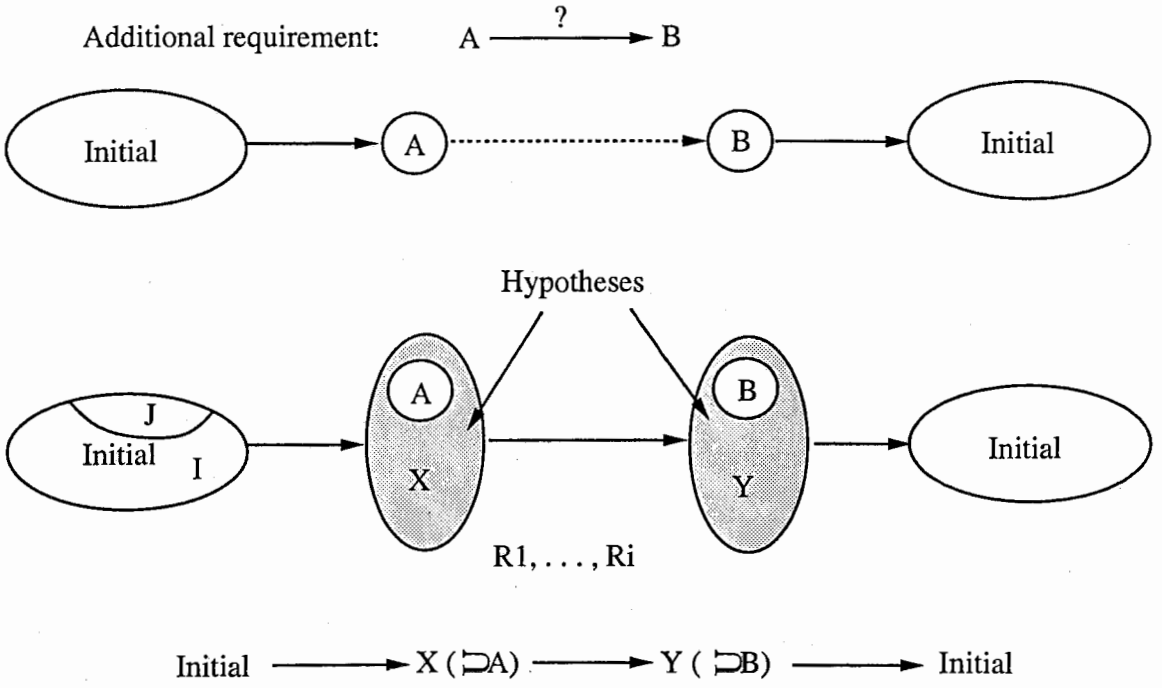


Figure 3.4: Hypothesis-based reasoning.

We show an example subjected to reachability analysis with hypothesis-based reasoning in order to make up for missing rules. Let us consider the next requirement.

A: dial-tone(a)
B: path(a,b)

Backward hypothesis-based reasoning is used to detect the necessary rules to make state B reachable from state A. States A and B are augmented by the hypothesis to state X and Y, respectively, so that state Y is reachable from state X.

X: dial-tone(a),idle(b),m-cfv(c,b),¬(idle(c))
Y: path(a,b),path(b,a),m-cfv(c,b),¬(idle(c))

"¬(idle(c))" in states X and Y is a constraint for the states, but not a part of them. Figure 3.5 illustrates the hypothesis-based reasoning for this example.

Assume that state 1 surrounded by a bold square is the original target state, and state 2 "ring-back(a,b),ringing(b,A)" is inferred by applying rule pots-5 after adding hypothesis "path(b,a)" to state 1. Then, state 1 results in state 4. Furthermore, state 3 is inferred by applying rule cfv-10 after adding hypothesis "m-cfv(c,b), ¬(idle(c))" to state 2. Then, state 4 results in state 6. Consistency is examined when a hypothesis is added to a state.

In this example we assume that X-A="idle(b),m-cfv(c,b),¬(idle(c))" is reachable independent of "dial-tone(a)". Then rules pots-5 and cfv-10 are the achieved rules. We note that we cannot obtain supplementary rules without hypothesis-based reasoning because "path(a,b)" is a part of a state of two terminals.

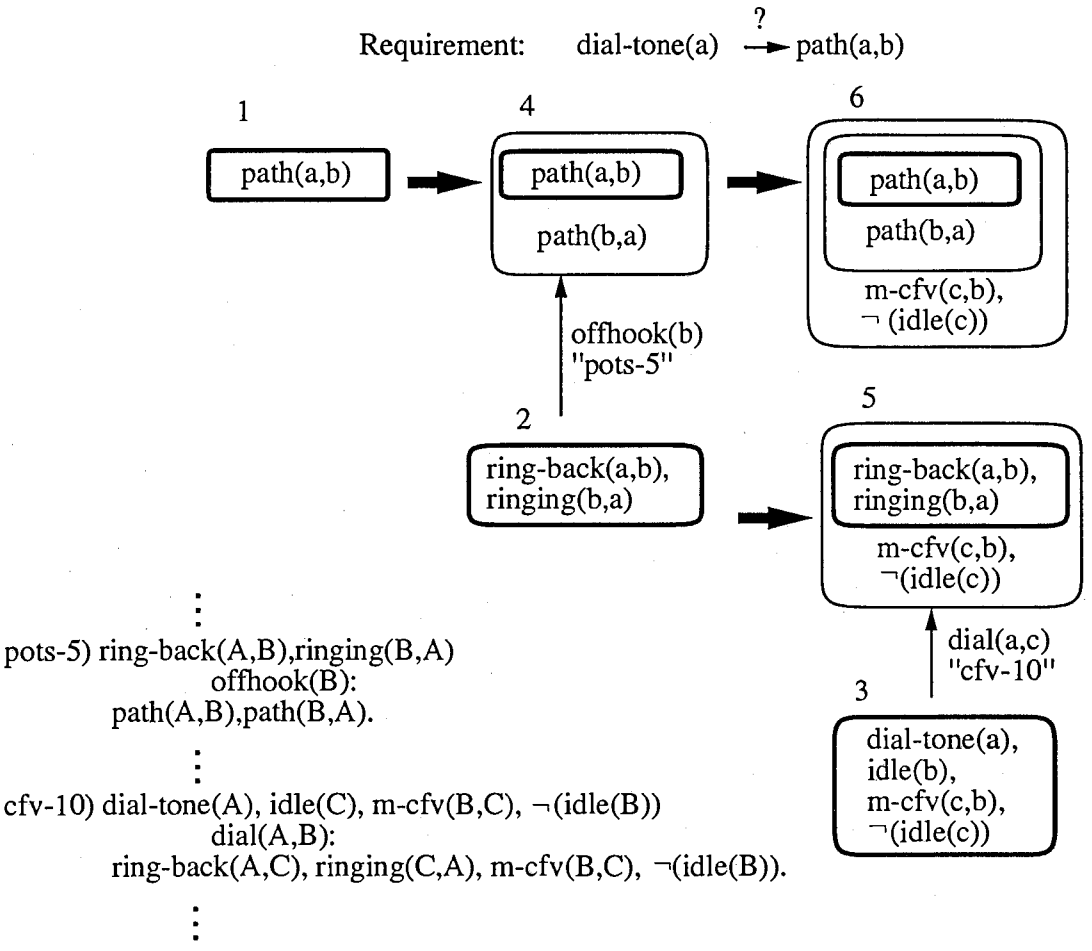


Figure 3.5: Example of hypothesis-based reasoning.

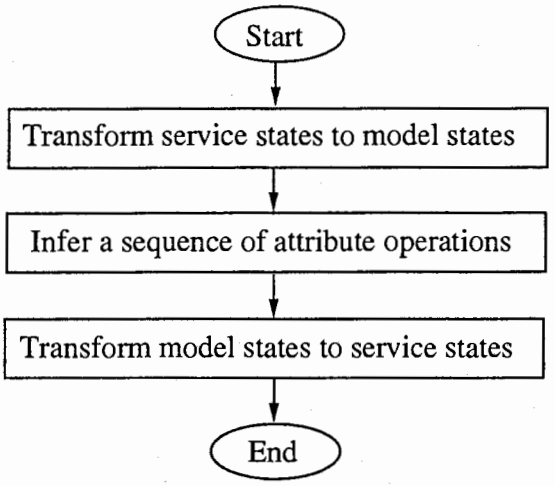


Figure 3.6: Model-based reasoning.

3.7 Generating New Rules

3.7.1 Domain Model

There is a case where existing rules are not sufficient to supplement missing rules. In this case we have to generate new rules. A domain model that is an abstraction of communication services is used to generate new rules.

Such a domain model consists of domain knowledge and a domain dictionary. The domain knowledge represents a communication service model expressing abstract communication services. This communication service model consists of attributes and attribute operations. The attributes characterize the communication services. The attribute operations have functions to examine and change the values of attributes. The domain dictionary provides functions for transformation between service specifications in STR and communication service models in the domain model.

New rule generation with the domain model involves the following three steps. An outline is illustrated in Fig. 3.6.

Step 1 Transform a detected deadlock state and a target state designated to be reached from it into states of a communication service model by using the domain dictionary.

Step 2 Perform inference on the communication service model to obtain a sequence of attribute operations that change the model state transformed from the deadlock state to the model state transformed from the target state.

Step 3 Transform the acquired attribute operations into a set of STR rules by using the domain dictionary.

We define attributes and attribute operations. First, we introduce a call element to define a communication service model for any services with primitive attributes and attribute operations. There are two types of call elements that show unary and binary relations among terminals. The former is called type 1 call element and the latter type 2 call element. A state

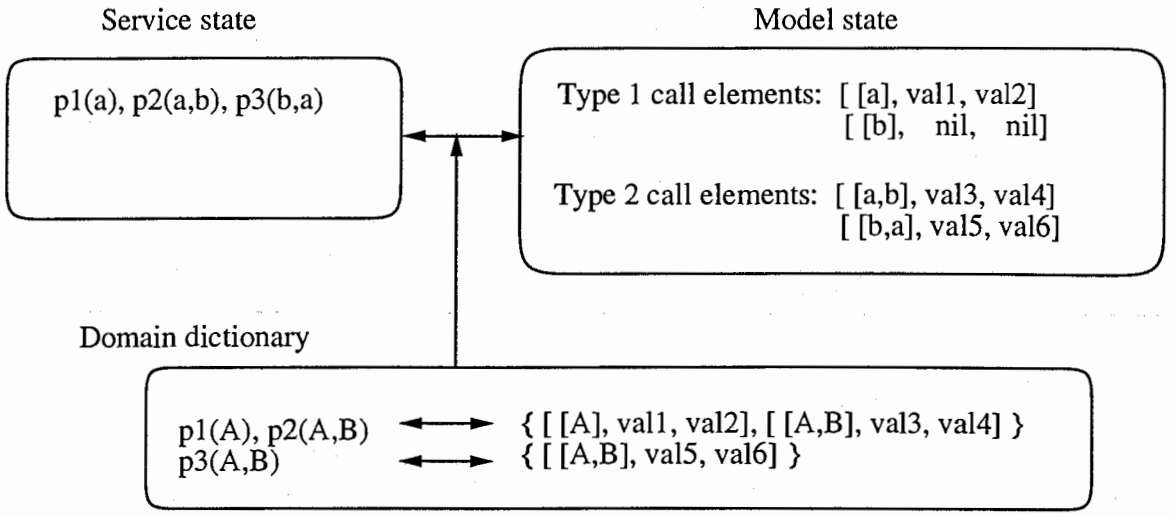


Figure 3.7: Transformation between service and model states.

of n terminals in the domain model is represented as a set of n type 1 call elements for all terminals and $n \times (n-1)$ type 2 call elements for all combinations among n terminals. Each type of call element is defined as a set of attributes. An attribute may have a value called an attribute value. The number of terminals that participate in a communication service is indefinite in general. A state in the domain model with a combination of call elements makes it possible to represent any communication service in a communication service model.

We call a state represented by STR a "service state", and a state of a communication service model a "model state". A model state is defined as a set of call elements, and each call element is defined as a set of attribute values. Every call element has identifiers called address attributes. There are two address attributes for the originating address and the terminating address. A type 1 call element has only an originating address. A type 2 call element, on the other hand, has both types of addresses. The originating address represents a terminal that a call element belongs to. A type 2 call element is identified by its originating and terminating addresses. When a certain service state is given, the set of call elements that defines the corresponding model state is automatically determined. An element of the domain dictionary defines a relation between a set of state primitives and a set of attributes of one or plural call elements. The transformation between service and model states is composed by the relations defined in the domain dictionary.

Figure 3.7 shows a transformation example.
Service state

"p1(a),p2(a,b),p3(b,a)"

corresponds to model state

Type 1 call elements : { [[a], val1, val2],
[[b], nil, nil] }
Type 2 call elements : { [[a,b], val3, val4],
[[b,a], val5, val6] }

This correspondence is obtained by the next relations.

$$\begin{array}{lcl} \text{p1(A),p2(A,B)} & \leftrightarrow & \{ \text{[[A], val1, val2], [[A,B], val3, val4]}\}, \\ \text{p3(A,B)} & \leftrightarrow & \{ \text{[[A,B], val5, val6]}\} \end{array}$$

A model state can be changed to another model state by modifying values of its attributes. This modification is performed by a sequence of attribute operations. There are two types of attribute operations; primary attribute operations and subordinate attribute operations. The difference between both types of operations is that the former have a property to determine the arguments of an event. A primary attribute operation consists of an application condition part, operation condition part, operation part and event argument part. A subordinate attribute operation consists of an application condition part, operation condition part and operation part

The application condition part screens applicable attribute operations for model states. An application condition is composed by a call element identifier and an attribute condition. The attribute condition is a logical formula with conjunction, disjunction and negation of value tests ($IF).

A call element identifier is used for screening call elements examined by the succeeding attribute condition.

Syntax: $CE(call element type, address 1, address 2)

A call element identifier has three arguments:the call element type, the first address attribute and the second address attribute. The call element type is either a type 1 call element, a type 2 call element or unspecified "-". The first address attribute always represents the originating address. The second address attribute is always unspecified for a type 1 call element and is a terminating address for a type 2 call element. "-" is used for specifying an arbitrary originating or terminating address. Note that we can use *oa* to represent the originating address of a screened call element in a context, and *ta* the terminating address.

An attribute condition is used for screening call elements that have already been screened by a call element identifier. We can specify concrete attributes and attribute values in an attribute condition.

Syntax: $IF(attribute name, attribute value)
$IF(attribute name, -)
$IF(attribute name)
$IF(attribute name, variable)

$IF(attribute name, attribute value): tests if the value of the attribute designated by "attribute name" is equal to "attribute value".

$IF(attribute name, -): tests if the attribute with "attribute name" is undefined.

$IF(attribute name): tests if the attribute has a value.

$IF(attribute name, variable): tests if the value of the attribute designated by "attribute name" is equal to the value of an attribute with the same "variable".

An operation condition is used for screening call elements from the selected ones by the application condition. The selected call elements are applied by the operations at the operation part. An operation condition is a logical formula with conjunction and negation of call element identifiers and attribute conditions.

Operations at the operation part are applied to call elements that are finally screened by the operation condition.

Syntax: $VALUE(attribute name, attribute value)
$VALUE(attribute name, -)
$VALUE(attribute name)

$VALUE(attribute name, attribute value): changes the attribute value of the attribute designated by "attribute name" to "attribute value".

$VALUE(attribute name, -): changes the value of the attribute designated by "attribute name" to "undefined".

$VALUE(attribute name): An initial value is set up for the attribute designated by "attribute name".

The event argument specifies a call element representing arguments of an event. The syntax is the same as that of the application condition except that no disjunctive formula can be described.

3.7.2 Reasoning with Domain Model

Rule generation method

We show a method of generating new STR rules when both the start and target states are given. The generated rules link from the start state to the target state. The next procedure generates missing rules.

Step 1 Augment the start state so that the set of call elements for the start state includes the set of call elements for the target state. Hereafter, the augmented start state is simply called the start state.

Step 2 Transform the start and target states into the sets of call elements according to the domain dictionary. The set of attribute values in the call elements of the start state is called the start model state, and that of the target state is called the target model state.

Step 3 Search for sequences of attribute operations with satisfiable application conditions from the start model state to a state including the target state. Here the inclusion between the two model states is defined as inclusion between their call elements. In this search we use the distance between two model states defined later.

Step 4 Transform the searched attribute operation sequences into sets of STR rules with the domain dictionary.

Distance

We define the distance between two model states. Let s and t be two model states. The number of attributes with different values in the common call elements of s and t is defined as the distance between s and t, and is denoted by d(s,t). Figure 3.8 shows two model states between which the distance is two.

Model reasoning

We show a procedure to search for attribute operation sequences from a start model state to a target model state. An obtained sequence is a sequence of primary attribute operations and subordinate ones. In what order an attribute operation is chosen is as follows. First, an attribute operation is chosen so that the distance between the operation applied model state and the target model state becomes short. This selection is called immediate reasoning. If there is no such attribute operation, another attribute operation is chosen such that the distance does

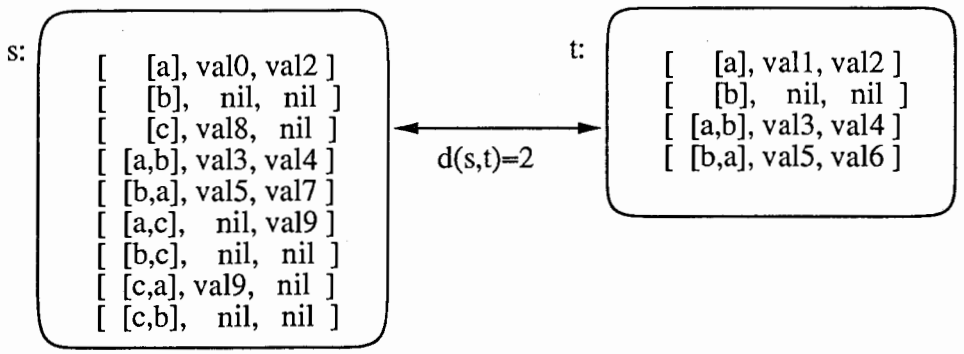


Figure 3.8: Distance between model states.

not change before and after an application of the operation. This selection is called detour reasoning. If there is no such attribute operation, the third choice is to choose an attribute operation such that the distance becomes long after an application of the operation. This is called devious reasoning. Figure 3.9 illustrates these three types of reasoning.

3.7.3 Example

We show an example to infer a service specification from a fragmentary requirement specification. Let us consider a requirement specification for a new service that will combine a call forwarding service and a call waiting service. Although a terminating call is forwarded like in an ordinary call forwarding service, functions belonging to a terminal are not transferred in this service. When a call waiting subscriber (B) sets up to transfer an incoming call to C, the incoming call to B is forwarded to C. If the call forwarding terminal is talking with someone else (D), however, the call to be forwarded is terminated to C as a call waiting service. If a user of terminal C flashes the terminal, the call between C and D is put on hold and C begins to talk with A.

Let the following requirement specification be given.

Start state: dial-tone(A),idle(B),m-cw(B),m-trans(B,C),path(C,D),path(D,C),
Target state: idle(B),m-cw(B),m-trans(B,C),
m-cw-transed(C),path(A,C),path(C,A),hold(C,D),hold(D,C).

Let the communication service model be an abstraction of a call waiting service, an ordinary call forwarding service and a function transfer service. We provide the next attributes for type 1 and type 2 call elements. For simplicity, we omit all other attributes necessary for representing actual communication services.

Type 1: [[orig], handset, tone, cw, ftrans, ftransed]
Type 2: [[orig, term], path, ring, trans]

Each attribute can have the following values.
Type 1 call element:

orig represents the originating address that the call element belongs to.

handset represents the state of a handset.

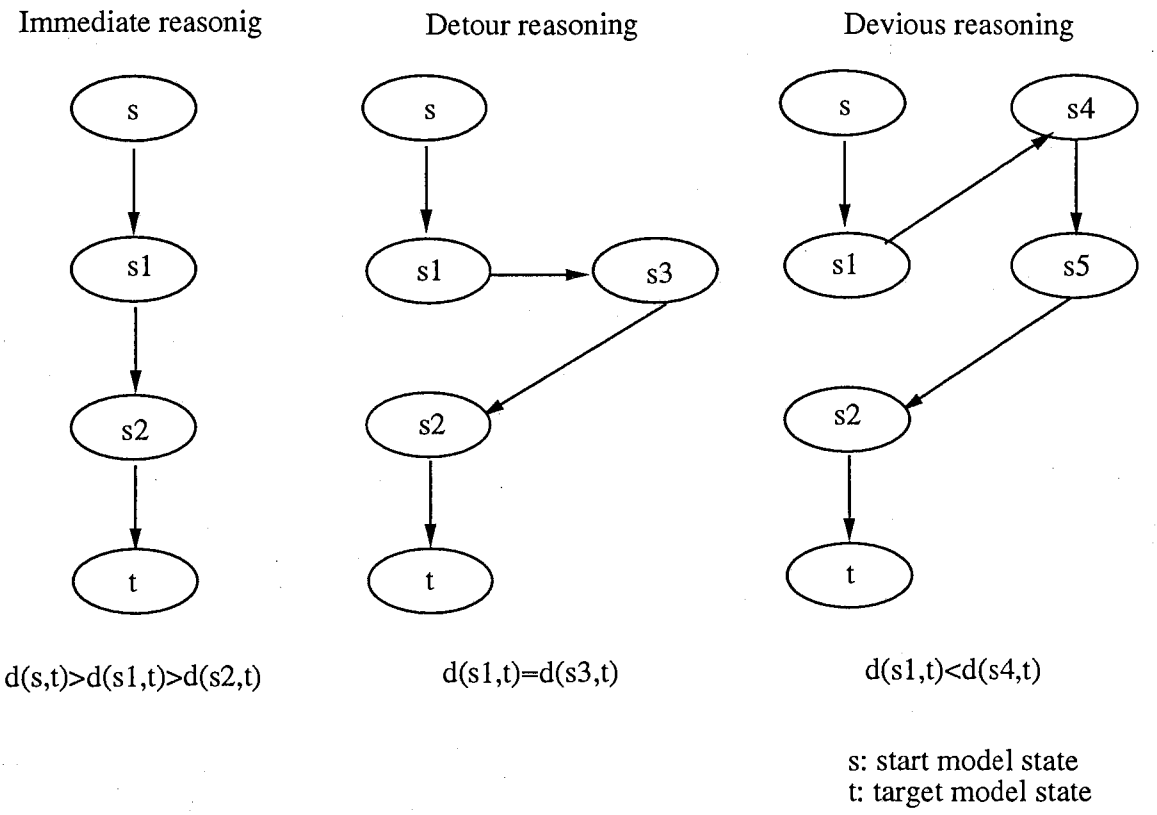


Figure 3.9: Various reasoning on a communication service model.

tone represents the tone state. It takes one of the following values: "in", "dial-tone" or "off". "in" represents the tone from a receiver, "dial-tone" is the dial-tone from a telephone and "off" means the no-tone state.

cw represents a registration state of the call waiting service.

ftrans represents the capability of transferring functions.

ftransed represents the transfered state of functions.

Type 2 call element:

orig and term represent the originating and terminating addresses of type 2 call elements.

path represents the talking state with the terminal designated by the terminating address.

ring represents the tone state from the terminal designated by the terminating address.

trans represents the state of function transfer. "on" represents the state that functions are transferred and "off" represents the state that functions are not transferred.

The following attribute operations are registered in the domain dictionary. The attribute operations with a label prefixed with PRIM are primary operations and that with the label SUB1 is a subordinate operation.

```
PRIM1:
  Apply_Condition:
    $CE(1,ad2,-):$IF(ftrans,on),$CE(2,ad2,ad3):$IF(trans,on);
  Operation:
    $CE(1,ad3,-):$IF(cw,off),$VALUE(cw,on),$VALUE(ftransed,on);
  Event_Condition:
    $CE(1,ad1,-):$IF(tone,dial-tone),$CE(2,ad2,ad3):$IF(trans,on),$CE(2,ad1,ad3);

SUB1:
  Apply_Condition:
    $CE(1,ad1,-):$IF(tone,dial-tone),$CE(2,ad2,ad3):$IF(trans,on);
  Operation:
    $CE(1,ad1,-):$VALUE(tone,off);
    $CE(2,ad1,ad3):$VALUE(ring,cw-rbt);
    $CE(2,ad3,ad1):$VALUE(ring,cw-rgt);

PRIM2:
  Apply_Condition:
    $CE(2,ad1,ad3):$IF(ring,cw-rbt),$CE(2,ad3,ad1):$IF(ring,cw-rgt),
    $CE(2,ad3,ad4):$IF(path,conn),$CE(2,ad4,ad3):$IF(path,conn);
  Operation:
    $CE(2,ad1,ad3):$IF(ring,cw-rbt),$VALUE(path,conn),$VALUE(ring,off);
    $CE(2,ad3,ad1):$IF(ring,cw-rgt),$VALUE(path,conn),$VALUE(ring,off);
    $CE(2,ad3,ad4):$IF(path,conn),$VALUE(path,hold);
    $CE(2,ad4,ad3):$IF(path,conn),$VALUE(path,hold);
```

```
Event_Condition:
  $CE(2,ad3,ad4):$IF(path,conn),
  $CE(2,ad4,ad3):$IF(path,conn),
  $CE(1,ad3);
```

The following correspondences are elements of the domain dictionary.

Service state	Model state
dial-tone(T1)	[[T1],off,dial-tone,-,-,-]
m-cw(T1)	[[T1],-,-,on,-,-]
m-trans(T1,T2)	[[T1,T2],-,-,on]
path(T1,T2)	[[T1],off,-,-,-], [[T1,T2],conn,-,-]
cw-ringing(T1,T2)	[[T1],off,off,-,-,-], [[T1,T2],-,cw-rgt,-]
cw-ring-back(T1,T2)	[[T1],off,off,-,-,-], [[T1,T2],-,cw-rbt,-]
hold(T1,T2)	[[T1],off,-,-,-], [[T1,T2],hold,-,-]
idle(T1)	[[T1],on,off,-,off,-]
m-cw-transed(T1)	[[T1],-,-,on,-,on]

Event	Attribute operation
PRIM1,SUB1	dial
PRIM2	flash

We show a reasoning result on the service model defined above. The underlined values show modified attribute values.

Start state: dial-tone(A),idle(B),m-cw(B),m-trans(B,C),path(C,D),path(D,C)

{ [[A],off,dial-tone,off,off,off],[[A,B],disc,off,off],[[A,C], disc,off,off],[[A,D],disc,off,off],
[[B],on,off,on,on,off],[[B,A],disc,off,off],[[B,C],disc, off,on],[[B,D],disc,off,off],
[[C],off,off,off,off,off],[[C,A],disc,off,off],[[C,B],disc, off,off],[[C,D],conn,off,off],
[[D],off,off,off,off,off],[[D,A],disc,off,off],[[D,B], disc,off,off],[[D,C],conn,off,off]}

Half-finished state:

{ [[A],off,dial-tone,off,off,off],[[A,B],disc,off,off],[[A,C], disc,off,off],[[A,D],disc,off,off],
[[B],on,off,on,on,off],[[B,A],disc,off,off],[[B,C],disc, off,on],[[B,D],disc,off,off],
[[C],off,off,<u>on</u>,off,<u>on</u>],[[C,A],disc, off,off],[[C,B],disc,off,off],[[C,D],conn,off,off],
[[D],off,off,off,off,off],[[D,A],disc,off,off],[[D,B],disc,off,off],[[D,C],conn,off,off]}

Intermediate state: idle(B),m-cw-transed(C),m-cw(B),m-trans(B,C),
path(C,D),path(D,C),cw-ring-back(A,C),cw-ringing(C,A)

{ [[A],off,<u>off</u>,off,off,off],[[A,B],disc,off,off],[[A,C], disc,<u>cw-rbt</u>,off],[[A,D],disc,off,off],
[[B],on,off,on,on,off],[[B,A],disc,off,off],[[B,C],disc, off,on],[[B,D],disc,off,off],
[[C],off,off,on,off,on],[[C,A],disc,<u>cw-rgt</u>, off],[[C,B],disc,off,off],[[C,D],conn,off,off],
[[D],off,off,off,off,off],[[D,A],disc,off,off],[[D,B],disc,off,off],[[D,C],conn,off,off]}

Target state: idle(B),m-cw-transed(C),m-cw(B),m-trans(B,C),
path(A,C),path(C,A),hold(C,D),hold(D,C)

{ [[A],off,off,off,off,off],[[A,B],disc,off,off],[[A,C], <u>conn</u>,<u>off</u>,off],[[A,D],disc,off,off],
[[B],on,off,on,on,off],[[B,A],disc,off,off],[[B,C],disc, off,on],[[B,D],disc,off,off],
[[C],off,off,on,off,on],[[C,A],<u>conn</u>, <u>off</u>,off],[[C,B],disc,off,off],[[C,D], <u>hold</u>,off,off],
[[D],off,off,off,off,off],[[D,A],disc,off,off],[[D,B],disc,off,off],[[D,C],<u>hold</u>,off,off]}

The following STR rules r1 and r2 are generated.
dial-tone(A), idle(B), m-cw(B), m-trans(B,C), path(C,D), path(D,C)

dial(A,B):
idle(B), m-cw-transed(C), m-cw(B), m-trans(B,C), cw-ring-back(A,C), cw-ringing(C,A), path(C,D), path(D,C).

cw-ring-back(A,C), cw-ringing(C,A), m-cw-transed(C), path(C,D), path(D,C)
flash(C):
path(A,C), path(C,A), hold(C,D), hold(D,C), m-cw-transed(C)

In the above example the domain dictionary is completely defined concerning the given requirement specification. However, there are cases in which transformations between service states and model states cannot be performed completely. In such cases, the transformations are assisted by a service designer.

3.8 Effectiveness and Limits

Here, we clarify the effectiveness and the limits of the proposed specification completion method, and describe future work. We have presented a requirements acquisition method from an incomplete requirement specification. Requirement specifications are described by a set of rules, and the acquired specification agrees with the requirements of the service designer. Most published works start with the premise that the requirements analysts are different from the users of the software systems. However, our method allows users to describe their own requirements with STR. The proposed method has the ability to correct wrong rules and to supplement rules for acquiring complete service specifications. If new rules need to be generated, the requirement specifications are converted to a communications service model that is an abstraction of the communications services. The proposed method has a limit, however. If a completely new requirement specification, i.e., a specification beyond a provided service model, is given, it becomes impossible to generate rules to supplement the incomplete requirement specifications. Future work includes providing a generic domain dictionary and generic domain knowledge.

Chapter 4

Protocol Synthesis for a Layered Architecture – Synthesizing Sequential Communication Protocols –

4.1 Graph Representation of Service Specifications

We use the graph representation of an STR rule to generate a communication protocol from service specifications described by STR. An STR rule can be represented by two graphs. The *initial graph* corresponds to the initial state and an event; the *next graph* corresponds to the next state. Both the initial graph and the next graph are called rule graphs. Figure 4.1 shows the graph representation of the rule described in Fig. 1.3.

A rule graph consists of a set of vertices and directed edges. Each vertex has its own name and some vertices have labels. A vertex is denoted by a circle. The name of the vertex is written in its circle, and the labels of the vertex are written near the circle. Each edge has labels that are written near it. An initial graph has a label that shows an event. A vertex designated by the first argument of this label is called an *event vertex.*

A vertex that has a label or an edge incident to it is called *labeled.* Other vertices are called *unlabeled.* For each vertex in an initial graph, there must be a path from the event vertex.

The global state for all of the terminals in a communication system is called a *system state*; the graph denoting the system state is called a *system graph.* An STR rule states that the system graph's subgraph that is isomorphic to the initial graph of the rule should be replaced by the next graph of the rule. Figure 4.2 illustrates how an STR rule is applied. If there are

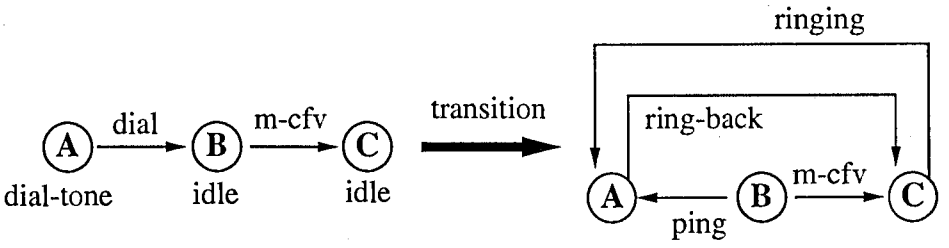


Figure 4.1: Graph representation of an STR rule.

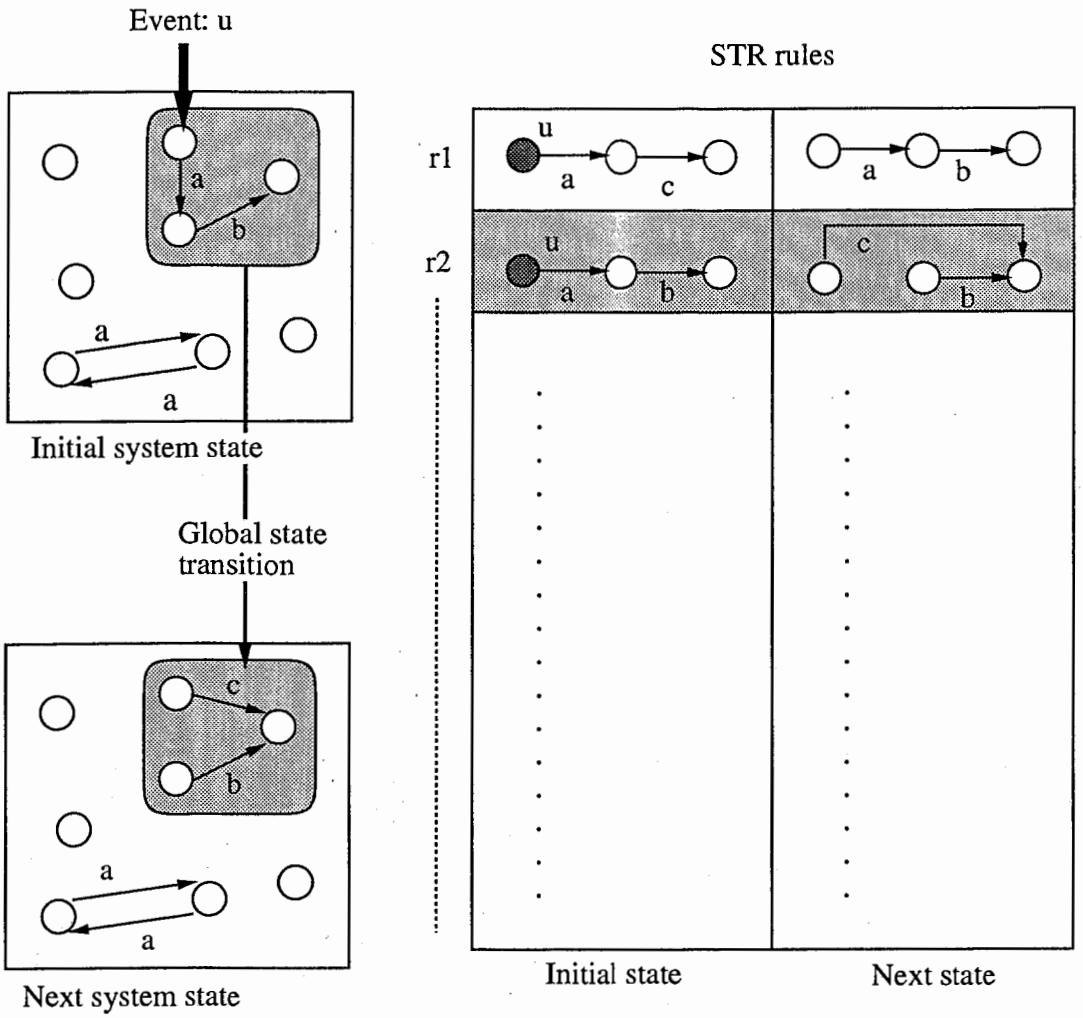


Figure 4.2: Application of STR rules.

multiple subgraphs isomorphic to the initial graph, an arbitrary maximal subgraph is selected.

4.2 Problem

Let $g = (V, E, v^0)$ be a rooted labeled directed graph that has a set of vertices V, a set of directed edges E and a root vertex v^0. For each vertex of the rooted directed graph, there is a path from v^0 to the vertex. The set of vertices of g is denoted by $V(g)$. The set of edges of g is denoted by $E(g)$. The root of g is denoted by $root(g)$. A labeled directed graph g is denoted by $g = (V, E)$ with a set of vertices V and a set of edges E.

Definition 1 (Subgraph isomorphism)
Let $g = (V, E)$ be a labeled directed graph and $g_1 = (V_1, E_1, v_1^0)$ be a rooted labeled directed graph. The graph g contains a subgraph $g' = (V', E', v^0)$ isomorphic to g_1 if and only if there exist subsets $V' \subset V$ and $E' \subset E$ such that $v^0 \in V'$, $|V'| = |V_1|$, $|E'| = |E_1|$, and there is a

one-to-one mapping $f : V_1 \longrightarrow V'$ *that satisfies the following conditions.*

$$f(v_1^0) = v^0$$
$$(x, y) \in E_1 \Longrightarrow (f(x), f(y)) \in E'$$
$$(f(x), f(y)) \in E' \Longrightarrow (x, y) \in E_1$$
$$\alpha(x) \subset \alpha(f(x))$$
$$\beta((x, y)) \subset \beta((f(x), f(y)))$$

where α *is a function to get a set of labels attached on a vertex and* β *is a function to get a set of labels attached on an edge.*

If g has a subgraph isomorphic to g_1, then we write $g_1 \sqsubset g$.

Definition 2 (Spanning path)
For a rooted labeled directed graph g*, a spanning path* $sp(g)$ *is defined as a path that satisfies the following conditions.*

1. *The vertices of* $sp(g)$ *make up the set of labeled vertices in* g.
2. *The root of* $sp(g)$ *is* $root(g)$.
3. *If* (u, v) *is an edge of* $sp(g)$*, then there are paths in* g *from* $root(g)$ *to* u *and from* $root(g)$ *to* v *such that every vertex of the paths except* v *appears before* u *in* $sp(g)$.

The problem to be solved is formally defined as follows.

Definition 3 (Problem)
Let $R = \{r_1, ..., r_n\}$ *be a set of STR rules. Let* $G = \{g_1, ..., g_n\}$ *be a set of initial graphs of* R. *Let* $G' = \{g'_1, ..., g'_n\}$ *be a set of next graphs of* R. *Let* $g = (V, E)$ *be a system graph.*

Find a pair of graphs $(sg(v^0, g), g_i)$ *such that* $sg(v^0, g)$ *is isomorphic to* $g_i \in G$*, and there is no graph isomorphic to* $g_k \in G$ *such that* g_i *is isomorphic to a subgraph of* g_k*. Then change* $sg(v^0, g)$ *to be isomorphic to* $g'_i \in G'$.

The following lemma is satisfied for subgraph isomorphism. We use this lemma to construct a distributed algorithm.

Lemma
Let $g = (V, E)$ be a labeled directed graph and $g_1 = (V_1, E_1, v_1^0)$ be a rooted labeled directed graph. Let p be a spanning path of g_1 with the set of edges $\{(u_1, u_2), \cdots, (u_{m-1}, u_m)\}$ where $u_1 = v_1^0$, and $V_1 = \{u_1, \cdots, u_m, u_{m+1}, \cdots, u_n\}$. The graph g contains a subgraph $g' = (V', E', v^0)$ isomorphic to g_1 if and only if $|V'| = |V_1|$, $|E'| = |E_1|$, and there is a mapping $f : V_1 \longrightarrow V'$ that satisfies the following conditions.

For any labeled vertices u_i, u_j in V_1 and unlabeled vertices u_k, u_l in V_1,
$f(u_1^0) = v^0$,
$u_i \neq u_j \Longrightarrow f(u_i) \neq f(u_j)$,
$f(u_i) \neq f(u_k)$,
$u_k \neq u_l \Longrightarrow f(u_k) \neq f(u_l)$,
$(u_i, u_j) \in E_1 \Longleftrightarrow (f(u_i), f(u_j)) \in E'$,
$\alpha(u_i) \subset \alpha(f(u_i))$
$\beta((u_i, u_j)) \subset \beta((f(u_i), f(u_j)))$
$\beta((u_i, u_k)) \subset \beta((f(u_i), f(u_k)))$

This lemma implies that we can determine the applicability of a rule by traversing along its spanning path.

Next we give the definition of numbering that enables an optimization of communication for querying if a set of initial graphs are included in a system graph. This numbering defines a spanning path in each initial graph.

Let $g(r_1)$, $\cdots$, $g(r_m)$ be initial graphs with event e. For a set of vertices V defined by $V(g(r_1))\cup\cdots\cup V(g(r_m))$ and a natural number n, let f: V $\longrightarrow\{1,\cdots,n\}\cup\{\lambda\}$ be a numbering function that satisfies the following six conditions:

1. If $v(\in V)$ is a labeled vertex, then $f(v)\in\{1,\cdots,n\}$; otherwise $f(v)=\lambda$.
2. $f(root(g(r_1)))=\cdots=f(root(g(r_m)))=1$, where $root(g(r_i))$ represents the root of $g(r_i)$.
3. For any two vertices $u,v(\in V(g(r_i)))$, $u\neq v$ implies $f(u)\neq f(v)$.
4. For any two vertices $u\in V(g(r_i))$, $v\in V(g(r_j))$, let V_1, V_2 be two sets of vertices on $path(root(g(r_i)),u)$ and $path(root(g(r_j)),v)$, where $path(root(g(r_k)),w)$ is a directed path from $root(g(r_k))$ to w in $g(r_k)$. Then, $f(u)=f(v)$ implies that $f(V_1)=f(V_2)$.
5. Let $u,v\in V(g(r_i))$, $w,x\in V(g(r_j))$, $uv\in E(g(r_i))$ and $wx\in E(g(r_j))$. If $f(u)=f(v)$ and $f(v)=f(x)$, then $\beta(uv)\subseteq\beta(wx)$ or $\beta(wx)\subseteq\beta(uv)$, where $\beta(yz)$ represents the label of an edge yz.
6. If $f(u)\neq f(v)$ and x is a vertex different from u and v, condition 3 above does not hold for an arbitrary function g such that $g(u)=g(v)$ and $g(x)=f(x)$.

Figure 4.3 shows an example of numbering to vertices in initial graphs $r1$, $\cdots$, $r4$.

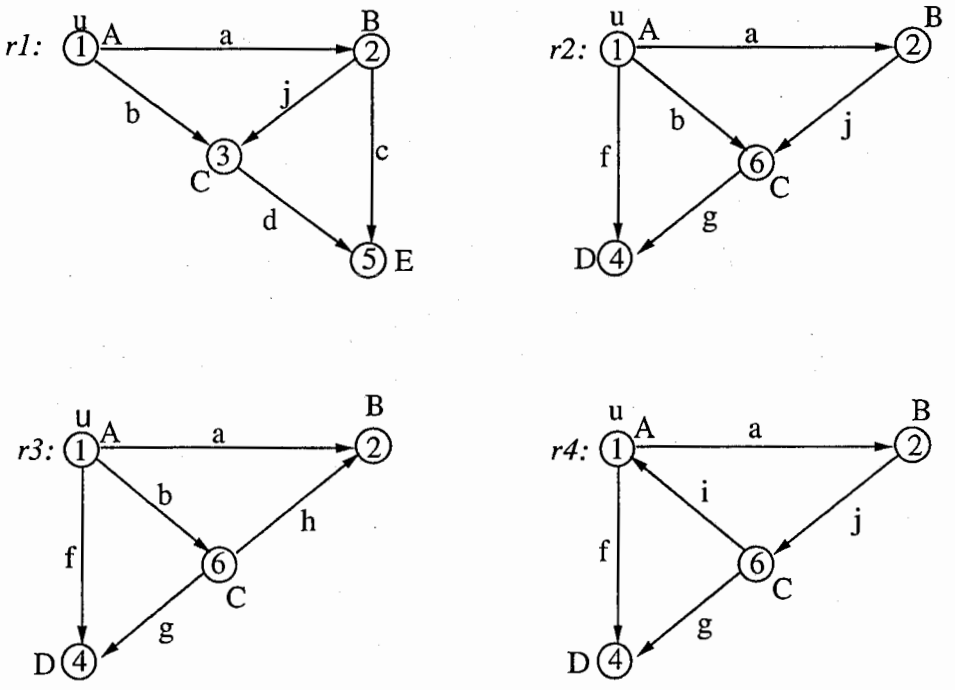


Figure 4.3: An example of numbering to vertices

The following properties hold for the above numbering.

Property 1 *Assume that an initial graph r is a subgraph of a system graph G. It can be decided that r is a subgraph of G by traversing vertices in the order of numbers assigned by the above numbering function.*

Property 2 *Let R be a set of initial graphs with the same event, and r_1 and r_2 be elements of R. Then, there is a numbering such that the set of numbers to the vertices of r_1 is a subset of numbers to the vertices of r_2.*

4.3 Definitions

Protocol synthesis involves generating a distributed algorithm to find an isomorphic subgraph in a system graph. Each protocol entity holds a local state of a terminal. Protocol synthesis is defined to derive protocol entity specifications with local states from communication service specifications described as global state transition rules.

When a protocol entity receives an input from a terminal or some other protocol entity, it determines a rule to be applied or sends another protocol entity a request to inquire about a surrounding global state determining an applicable rule. This communication for state inquiry is performed sequentially. Finally, a rule is determined. The rule is informed about the protocol entities that have had inquires about their states. Then each protocol entity is made to change its state according to the rule.

A spanning path is determined for each rule. The communication for state inquiry goes along a spanning path. If there are isomorphic subgraphs in multiple initial graphs, paths are determined at the same time regardless of whether a system graph contains a subgraph isomorphic to the subgraphs. This optimizes the communication. For this purpose, we utilize the spanning path of a tree generated by overlapping spanning trees of the initial graphs of an event. This spanning path is called a provisional communication path.

A synthesized protocol communicates along provisional communication paths. We give the following definitions. Note that sometimes a vertex and a process are used for the same meaning.

Let C(e) denote a set of initial graphs with an event e.

Rule inclusion graph Let s be a connected subgraph that has an event vertex and is isomorphic to subgraphs of more than one element in C(e). Let D(s) denote the set of elements of C(e) that have s as their subgraphs. An ordered pair $< s$, D(s) $>$ is a vertex of the rule inclusion graph of C(e) if a graph s' generated from s by adding an edge is a subgraph of an element of C(e), and D(s) $\neq$ D(s'). An ordered pair $< r$, D(r) $>$ is also a vertex of the rule inclusion graph of C(e) if $r \in$C(e). There is an edge from $< s$, D(s) $>$ to $< t$, D(t) $>$ iff (1) $s \sqsubset t$, and (2) there is no other element $< u$, D(u) $>$ such that $s \sqsubset u$ and $u \sqsubset t$.

Rule inclusion graphs are used for determining in what order the inquiring rules should be applied. If there is an edge from $< s$, D(s) $>$ to $< t$, D(t) $>$ in a rule inclusion graph, then graph s is examined before graph t regardless of whether they are included in a system graph.

Figure 4.4 gives initial graph examples. Figure 4.5 shows the rule inclusion graph for the set of initial graphs in Fig. 4.4. Rule graph $r1$ is the maximum subgraph common to rule graphs $r2$ and $r3$, and the maximum subgraph of rule graph $r4$ is rule graph $r2$. Consequently, there are four vertices $< r1, \{ r1, r2, r3, r4 \} >$, $< r2, \{ r2, r4 \} >$, $< r3, \{ r3 \} >$, and $< r4, \{ r4 \} >$ in this rule inclusion graph.

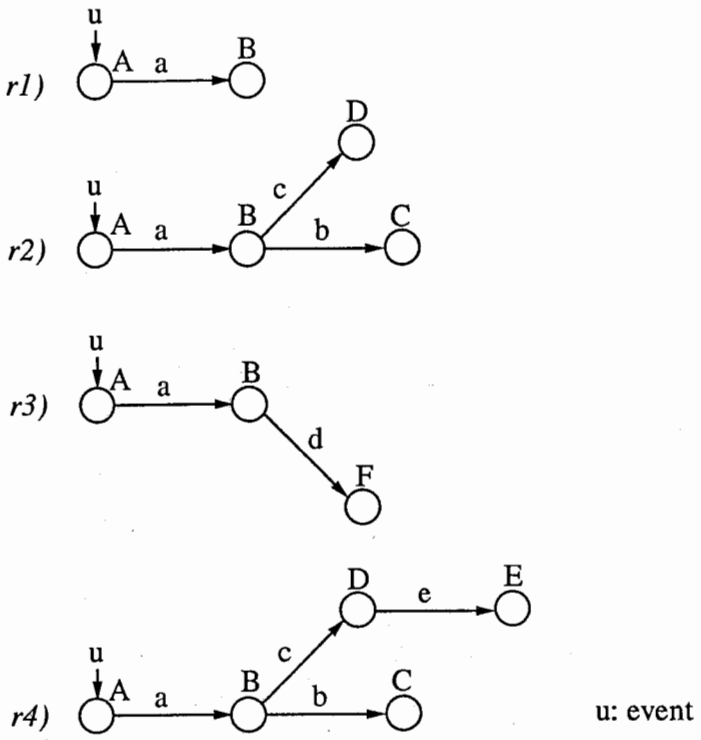


Figure 4.4: Examples of initial graphs.

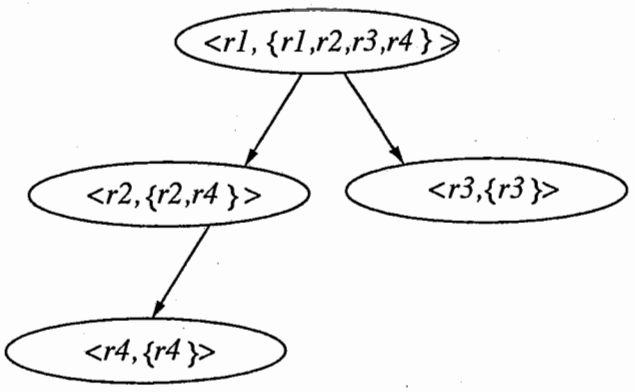


Figure 4.5: Example of a rule inclusion graph.

Rule covering tree Let $g_r = (v_r, e_r, v^0)$ be the initial graph of a rule r. Let $t_r = (w_r, f_r, v^0)$ be the spanning tree of g_r. A rule covering tree c_r is defined as follows:

The set of vertices of c_r is v_r, and $root(c_r) = v^0$.

Edges from the same vertex are arranged clockwise in lexicographic order of edge labels. The set of edges of c_r is the union of e_r and the edges satisfying the following two conditions:

1. If a leaf vertex of t_r has a label, there exists an edge whose initial vertex is the leaf vertex of t_r.

2. An edge ab of c_r has a label of an ordered pair. The first element of the label is the label of a in v_r, and the second element is the label of ab in e_r.

A rule covering tree is uniquely determined for a spanning tree of an initial rule graph. A rule covering tree must satisfy the following two conditions to define a rule overlapping tree. Let s be the spanning tree of the initial graph for rule r, and t a rule covering tree generated from s.

(1) s is an unlabeled subgraph of the initial graph for rule r.

(2) Let u be a vertex of s. Assume $g \sqsubset h$ for the two vertices g and h in the rule inclusion graph that has a vertex whose first element is r. If u is a vertex of h, and u is not a vertex of g, then an edge from one of the vertices in g to u is included.

Figure 4.6 shows rule covering trees for the initial graphs in Fig. 4.4.

Rule overlapping tree A rule overlapping tree for $C(e)$ is generated by overlapping graphs in $C(e)$. The tree satisfies the following conditions.

1. All the roots of the initial graphs in $C(e)$ are overlapped.

2. For each element g in $C(e)$, there is a subgraph isomorphic to g.

3. If vertices u and v in rule covering trees s and t, respectively, are overlapped, then every vertex u' between $root(s)$ and u is overlapped by the vertex v' in t satisfying the conditions that its depth is the same as that of u' and v' is a vertex between $root(t)$ and v (Fig. 4.7).

4. If edges e and f are overlapped, then the labels of e and f are the same or those of one are included in the other.

5. The overlapping is performed in lexicographic order of the labels of edges with the same initial vertex.

A rule overlapping tree is used for examining at the same time whether the common subgraphs in multiple rule graphs are included in a system graph.

Provisional communication path The provisional communication path for $C(e)$ is defined as a path whose vertices are vertices in the rule overlapping tree of $C(e)$ such that:

1. The initial vertex is the event vertex.

2. Let g and h be two graphs constituting the first elements of two vertices in the rule inclusion graph for $C(e)$. If $g \sqsubset h$, u is a vertex in both g and h; if v is not a vertex of g but a vertex of h, then u appears before v.

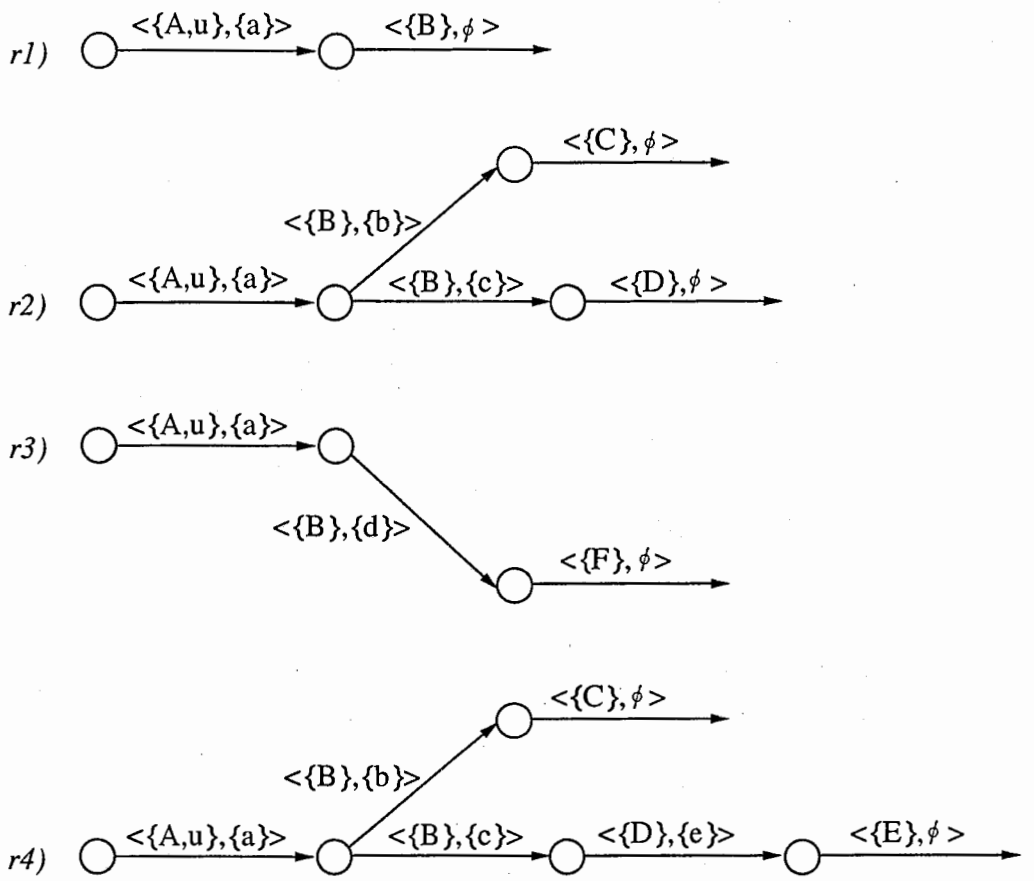


Figure 4.6: Examples of rule covering trees.

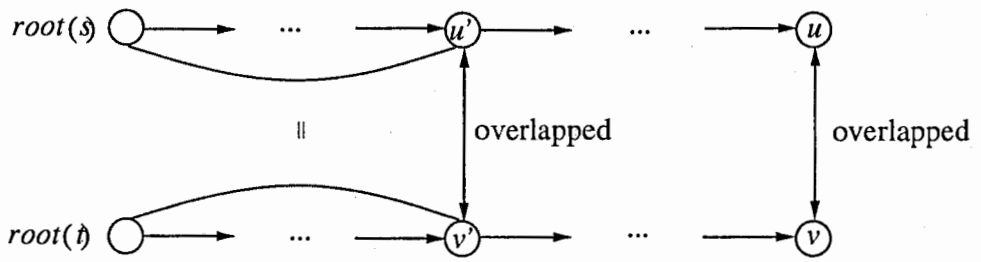


Figure 4.7: Overlapping of vertices.

3. Let h be a graph corresponding to a vertex in the rule overlapping graph for C(e). For any graph g corresponding to a vertex in the rule inclusion graph for C(e) such that $g \sqsubset h$, u and v are assumed to be vertices in h, but not in g. In h, if u is a vertex in the path from the event vertex to v, or there is a vertex w in the path from the event vertex to u such that w is in the path from the event vertex to v and w is not a vertex in g, then u appears before v.

4. A unique label is attached to each edge.

We note that the vertex sequence of the provisional communication path for C(e) includes the sequence of a spanning path for each element of C(e) as a subsequence. This implies that a state inquiry along a provisional communication path may determine a rule to be applied.

Inquiry message An inquiry message is a unique message for each edge between adjacent vertices in a provisional communication path. In the synthesized protocol, an inquiry message has the following information in addition to its message name.

Actual communication path An actual communication path is a provisional communication path whose vertices are actual process identifiers to be inquired. Each process decides which process to send an inquiry message by using this information.

Temporary decided rule The rule with the highest priority among the rules satisfying their rule application conditions, i.e., their initial graphs are included in the system graph.

Rule candidates The remaining rules to be checked for applicability. When a process receives an inquiry message it screens rule candidates included in a received inquiry message by checking its local state.

Connection information Process information necessary for connection tests.

Visited processes A sequence of visited processes after an event occurred.

Temporary process sequence The process sequence for the current temporary decided rule.

Branches A branch consists of a process identifier and an inquiry message. When a process receives an inquiry message, the process compares its state with a subgraph of the rule overlapping tree corresponding to the inquiry message, in order to obtain a new inquiry message and branches (i.e., actual communication paths). This subgraph is the intersection of the rule overlapping tree and rule covering trees corresponding to rules contained in the rule candidates of the inquiry message. If plural actual communication paths to be inquired are obtained, the remaining actual communication paths except arbitrary ones are stored as elements of inquiry messages in branches. Process identifiers are determined as the processes to which the obtained inquiry messages are to be sent.

A provisional communication path shows inquiry messages used for examining whether rule graphs are included in a system graph, and their communication paths. An inquiry message usually implies multiple rules to be checked as to whether they are included in a system graph.

Figure 4.8 shows the rule overlapping tree for the initial graphs in Fig. 4.4 and its provisional communication path.

Response message A message denoting whether to apply a rule or nothing at all. When a rule is indicated, it contains information on which process the received message corresponds to.

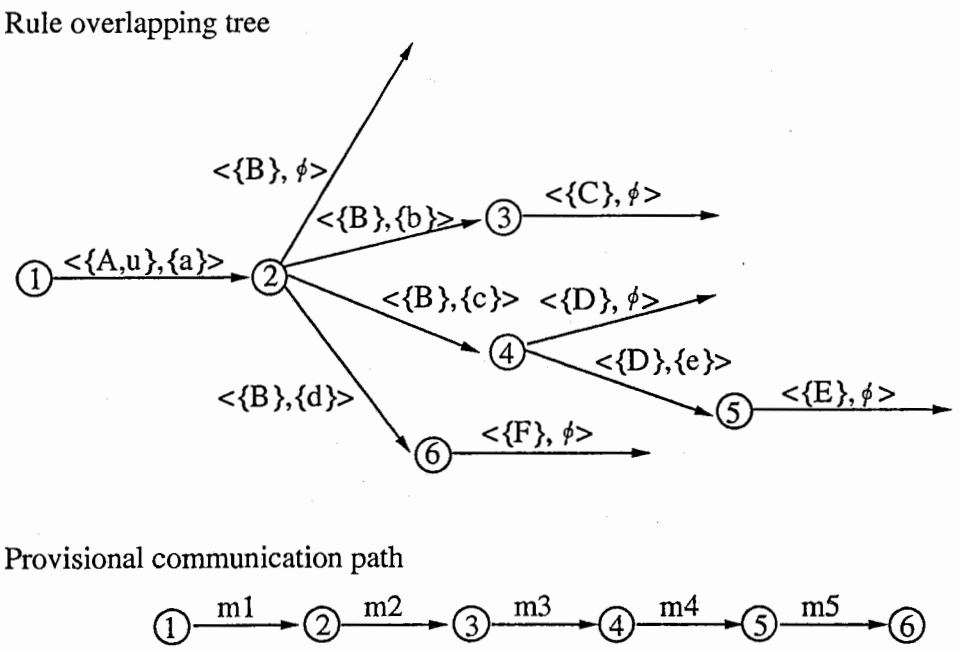


Figure 4.8: Example of a rule overlapping tree.

Figures 4.9 and 4.10 show actual communication paths when the graphs are included in a system graph. The dotted arrows represent communications between processes and the inquiry messages attached to them.

In the example of Fig. 4.9, inquiry messages are communicated along a provisional communication path. Process A sends B a message m1($r1$, $r2$, $r3$, $r4$) that implies rules $r1$, $r2$, $r3$ and $r4$ are rule candidates. When B receives it, B determines that rule $r1$ is included in the system graph, and rules $r2$, $r3$, and $r4$ are to be checked for their applicability. Then B sends C a message m2($r2$, $r3$, $r4$). When C receives this message, C sends m3($r2$, $r3$, $r4$) to D which is included in the received message as an actual communication path. In the end E receives a message m4($r3$, $r4$) and determines that rule $r4$ is included in the system graph. Since $r4$ is not included in any other initial graph, E determines that rule $r4$ is to be applied. Then E sends processes A, B, C, and D a response message indicating that each process will change its state according to rule $r4$. E also changes its state.

In the example of Fig. 4.10, inquiry message are communicated along a provisional communication path but some intermediate processes are skipped. When B receives a message m1($r1$, $r2$, $r3$, $r4$), B determines that rule $r1$ is a temporary determined rule, rules $r2$ and $r4$ are not included in the system graph, and rule $r3$ is to be checked as to its applicability. In the end F determines rule $r3$ is the rule to be applied.

Figure 4.11 shows the rule overlapping tree and its provisional communication path for the non-tree initial graphs in Fig. 4.3. Figure 4.12 illustrates provisional communication paths and inquiry messages for the rules in Fig. 4.3.

State transition segment One or two state transition segments are generated for each labeled vertex in the initial graph of a rule.

- When a vertex is the last vertex in a provisional communication path, a state tran-

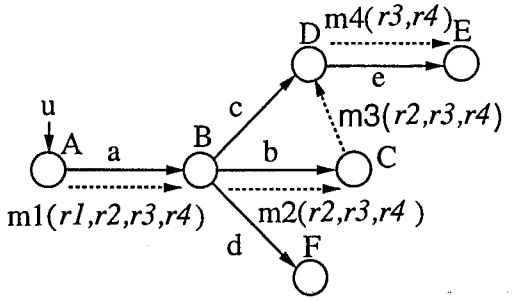


Figure 4.9: Example of communications along a provisional communication path.

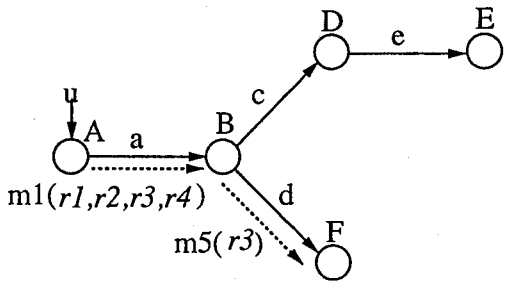


Figure 4.10: Example of communications skipping intermediate processes.

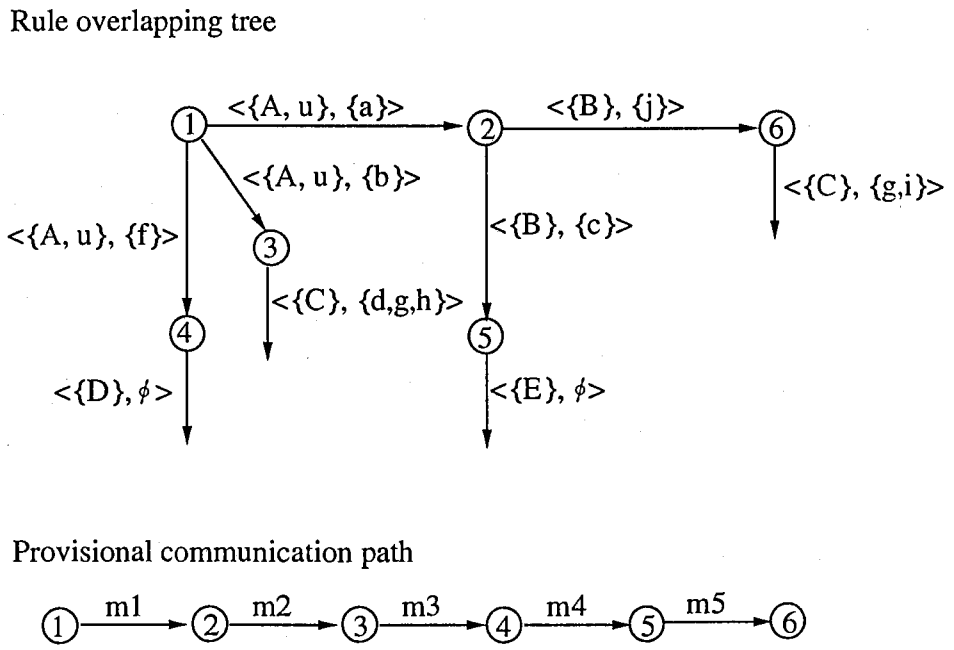


Figure 4.11: Example of a rule overlapping tree for complicated rules.

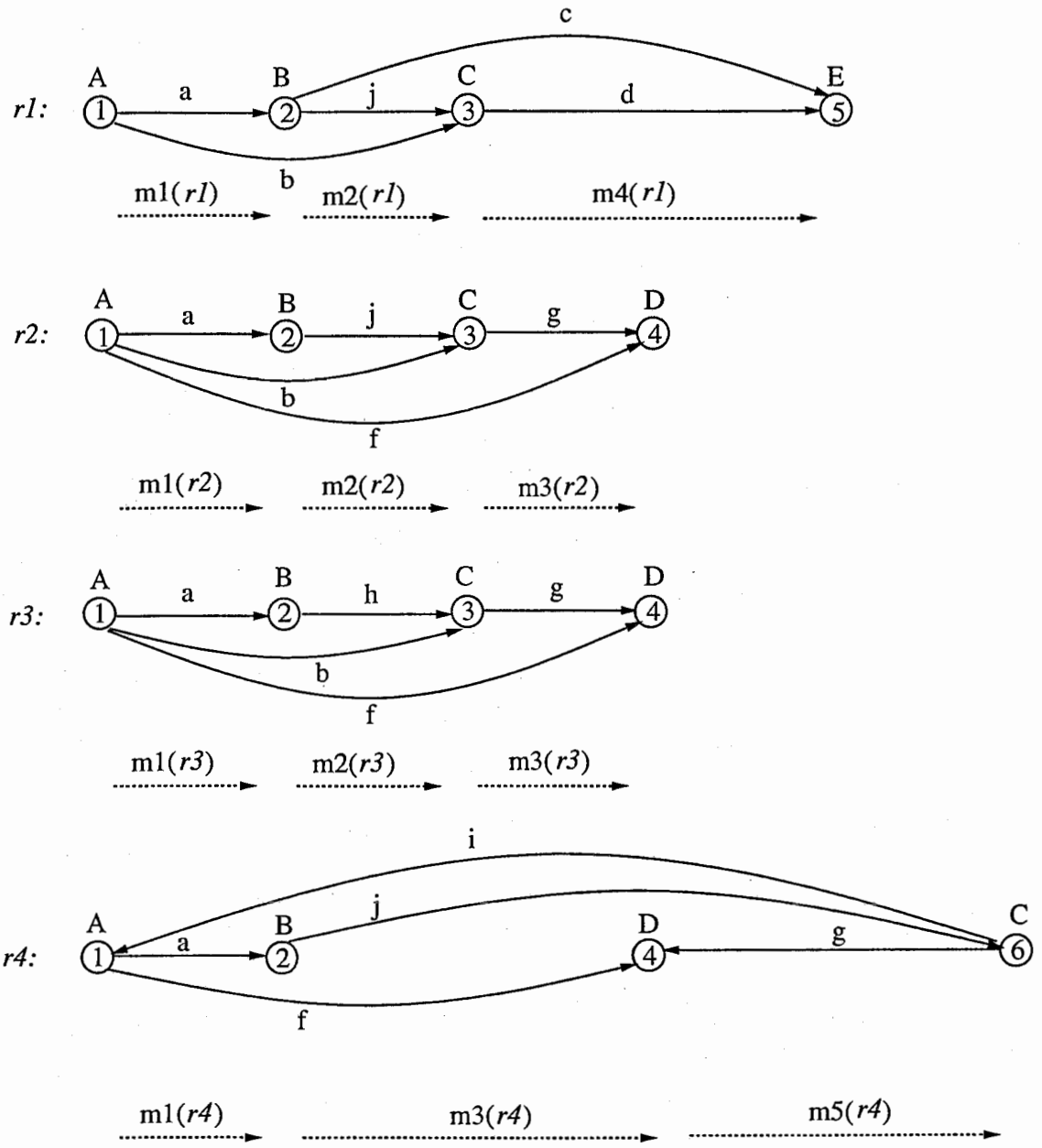


Figure 4.12: Communications for identifying rules in Fig. 4.3.

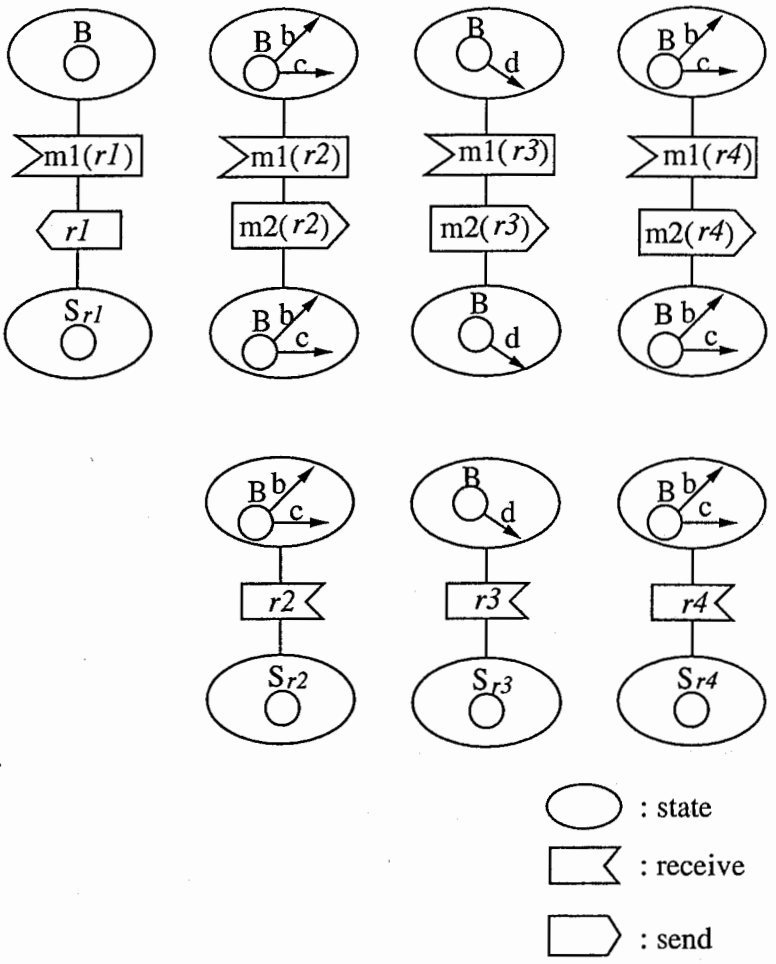


Figure 4.13: State transition segments.

sition segment is generated such that it receives a predetermined inquiry message in the provisional communication path; it sends a response message and changes to the next state determined by a rule.

- When a vertex is not the last vertex in a provisional communication path, two separate state transition segments are generated. One segment receives a predetermined inquiry message in the provisional communication path, sends another inquiry message along the provisional communication path, and then changes to the next state. The other segment receives a response message that includes a determined rule and changes to the next state.

Connection test A connection test is used to identify graphs that include plural vertices in a provisional communication path. Let i and j be two vertices in a provisional communication path, and i is nearer to the event vertex than j. The connection test examines if the edges from i excluding the edge in the provisional communication path are connected to vertices j or vertices connected by edges from j.

Figure 4.13 shows state transition segments of vertex B obtained from messages in the provisional communication path. Connection tests are omitted in this figure.

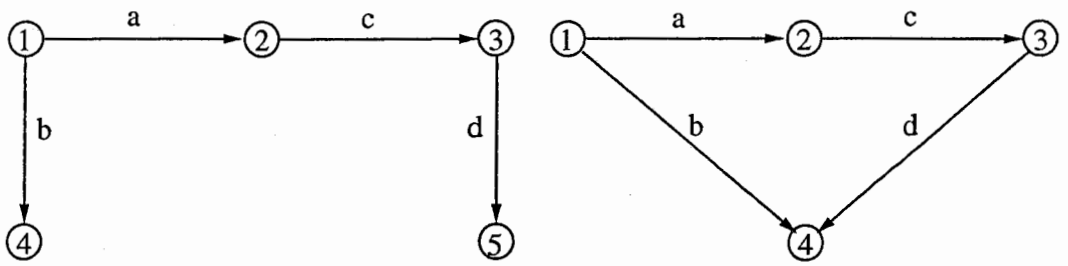


Figure 4.14: Two graphs requiring a connection test for identification.

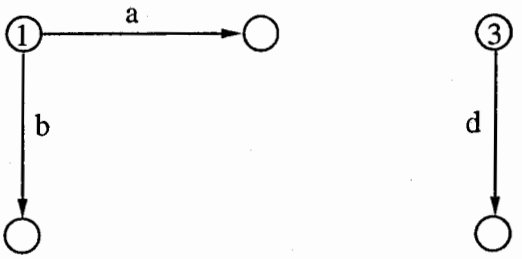


Figure 4.15: Subgraphs identifiable with local states.

Figure 4.14 shows two initial graphs that need a connection test to distinguish them. The provisional communication path in these two initial graphs is the path 1,2,3. Figure 4.15 shows subgraphs that can be identified by processes 1 and 3. The two initial graphs in Fig. 4.14 are distinguished when process 3 receives a message that edge b is connected to process 4.

4.4 Protocol Synthesis Algorithm

The protocol synthesis algorithm consists of the following six steps.

(1) Classification of rule graphs

Classify rules into sets of rules for the same event. Let C(e) be the set of a rule graph with an event e. For each rule graph set we initially apply steps 2 to 5; finally, step 6 is applied.

(2) Generation of rule inclusion graph

Generate a rule inclusion graph for each rule graph set.

(3) Generation of rule overlapping tree

Generate a rule overlapping tree.

(4) Determination of provisional communication path and inquiry messages

Determine a provisional communication path and inquiry messages from the rule overlapping tree.

(5) Generation of state transition segments

Generate state transition segments for each vertex of a rule covering tree corresponding to each rule r. The initial state of a state transition segment is the first element of the label attached to an edge incident from the vertex in the rule covering tree.

(6) Synthesis of process specification

Synthesize a state transition segment from the initial state until no new state is generated. Assume a new state s is generated. Collect all state transition segments whose initial state is included in s as a subgraph. The collected state transition segments are synthesized as follows.

- If two kinds of state transition segments correspond to the final vertex of an initial graph in the provisional communication path, a middle vertex is included, and their received messages are the same, the following synthesis occurs.

  The next states of these state transition segments are changed to the same states as their initial states. In the synthesized process specification, the rule candidates of the inquiry message to be sent are the intersection of the rule candidates of the received inquiry message and the set of rules whose state transition segments are synthesized. Other information is obtained as described in the above preliminaries.

  - When the resulting rule candidate set is not empty, the synthesized process sends the obtained inquiry message to the nearest process in the provisional communication path following the rules contained in the inquiry message.
  - When the resulting rule candidate set is empty and the branches in the received message are empty, two possible conditions exist: if the temporary decided rule is contained in the received inquiry message, the rule should be applied; if the temporary decided rule is not contained, a special response message is sent to all the visited processes.
  - When the resulting rule candidate set is empty but the branches in the received message are not empty, the process sends an inquiry message to find a more superior rule than the temporary decided rule.

- If the above does not occur, the following happens.

  The collected state transition segments are synthesized as they are. Each state transition segment is synthesized as described above. Figure 4.16 illustrates how collected state transition segments are synthesized into a process specification.

In the synthesized protocol specification, the rule candidates are screened as communication progresses, and then a rule to be applied is determined. Once a rule is determined to be applied at a process, the process sends response messages indicating the determined rule to the visited processes. There are two types of response messages: a message indicating a rule and no rule to be applied. A process that receives a response message changes to the state designated by the state transition segment corresponding to the determined rule. A process that receives a no rule message returns to the state prior to receiving the inquiry message.

4.5 Example

We explain an example of process specification generation. Figure 4.17 expresses a service specification. In this service we need two terminals: "data sender terminal" and "data receiver terminal". The data sender terminal starts data transmission by an event "start" when both the sender and the receiver which is specified by "start" are in the state "idle". The sender can always stop sending data by an event "stop". The receiver can always request the sender to pause sending data by an event "pause" and to resume the sending of data by an event

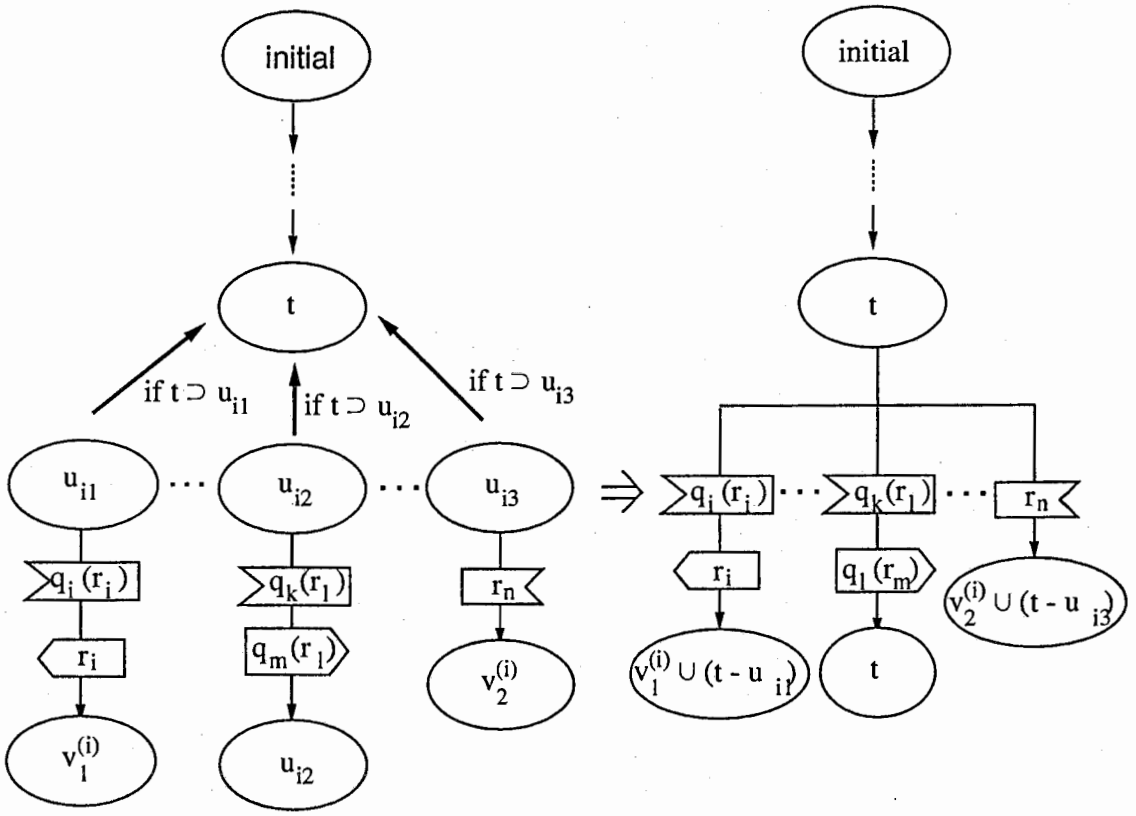


Figure 4.16: Synthesis of state transition segments.

```
r1) idle(A),idle(B) start(A,B):
          sending(A,B),receiving(B,A).
r2) sending(A,B),receiving(B,A) stop(A):
          idle(A),idle(B).
r3) receiving(A,B),sending(B,A) pause(A):
          r-wait(A,B),s-wait(B,A).
r4) r-wait(A,B),s-wait(B,A) resume(A):
          receiving(A,B),sending(B,A).
```

Figure 4.17: STR description for data sending protocol.

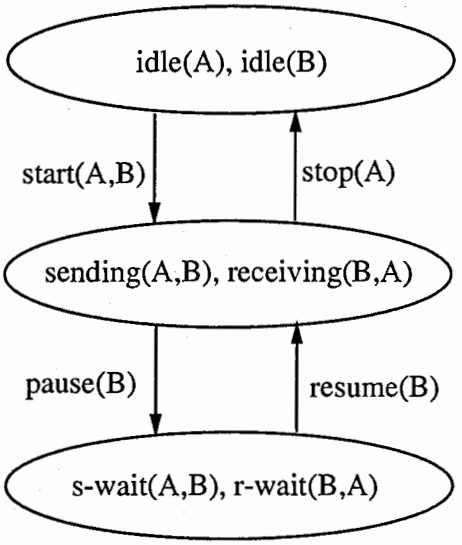


Figure 4.18: Global state transition diagram for data sending protocol with pause function.

"resume". Figure 4.18 shows a global state transition diagram for this data sending protocol with pause function.

We show the graph representation for the service specification in Fig. 4.17.

In this example each rule makes a rule overlapping tree. Using this graph representation and the rule overlapping trees, we can get state transition segments which are parts of the objective protocol entity specification. Figure 4.20 shows state transition segments for the STR rules in Fig. 4.17. The messages used in inter-process communication are generated from the provisional communication paths.

Figure 4.21 shows a protocol entity specification synthesized from state transition segments in Fig. 4.20. In Fig. 4.21 the messages "m1", "m2", "m3", and "m4" represent request messages, and the messages "r1", "r2", "r3", and "r4" represent response messages. In this specification "norule" send and receive are omitted. "Norule" is a special response message to indicate that there is no rule to be applied. In the generated protocol entity specification, when the protocol entity receives an unexpected request message, the protocol entity is assumed to send the message "norule" to the sender of the request message.

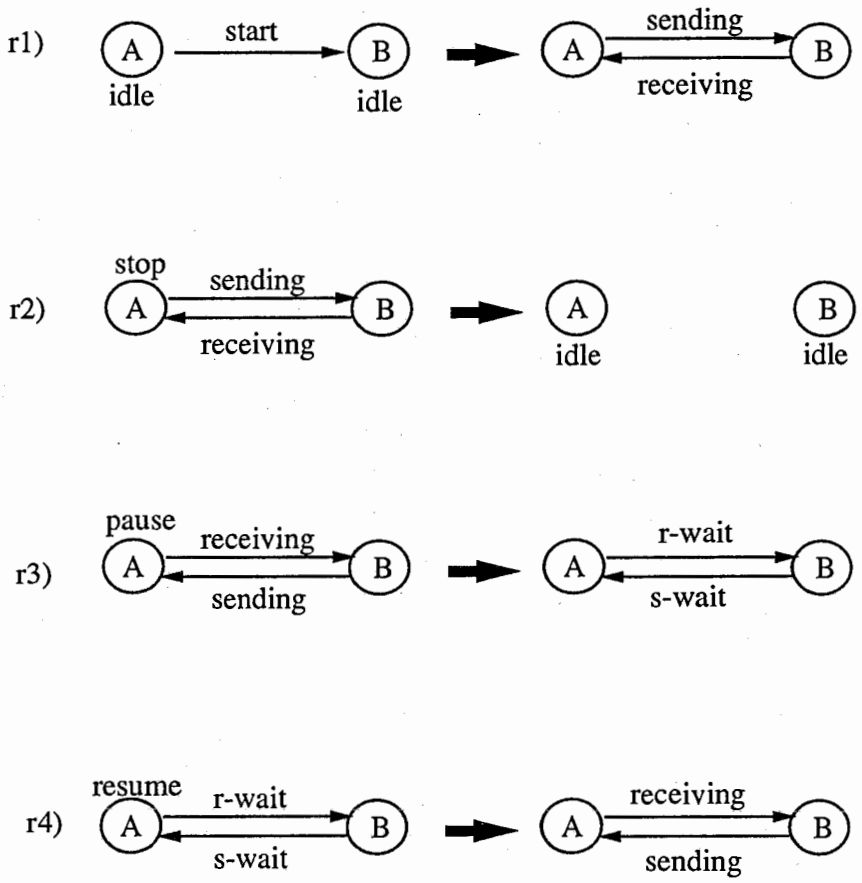


Figure 4.19: Graph representations of STR descriptions for data sending protocol.

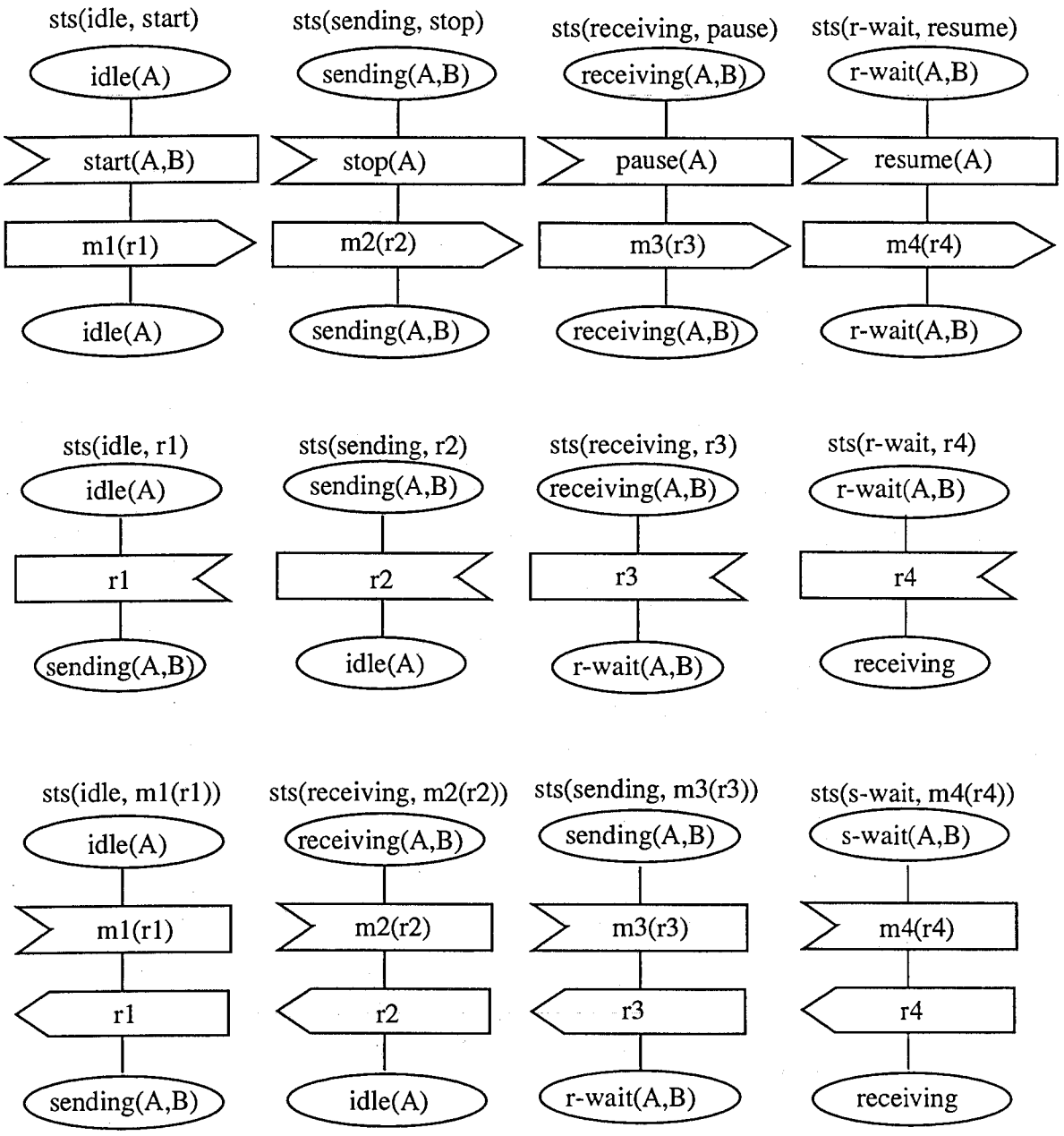


Figure 4.20: State transition segments for data communication protocol.

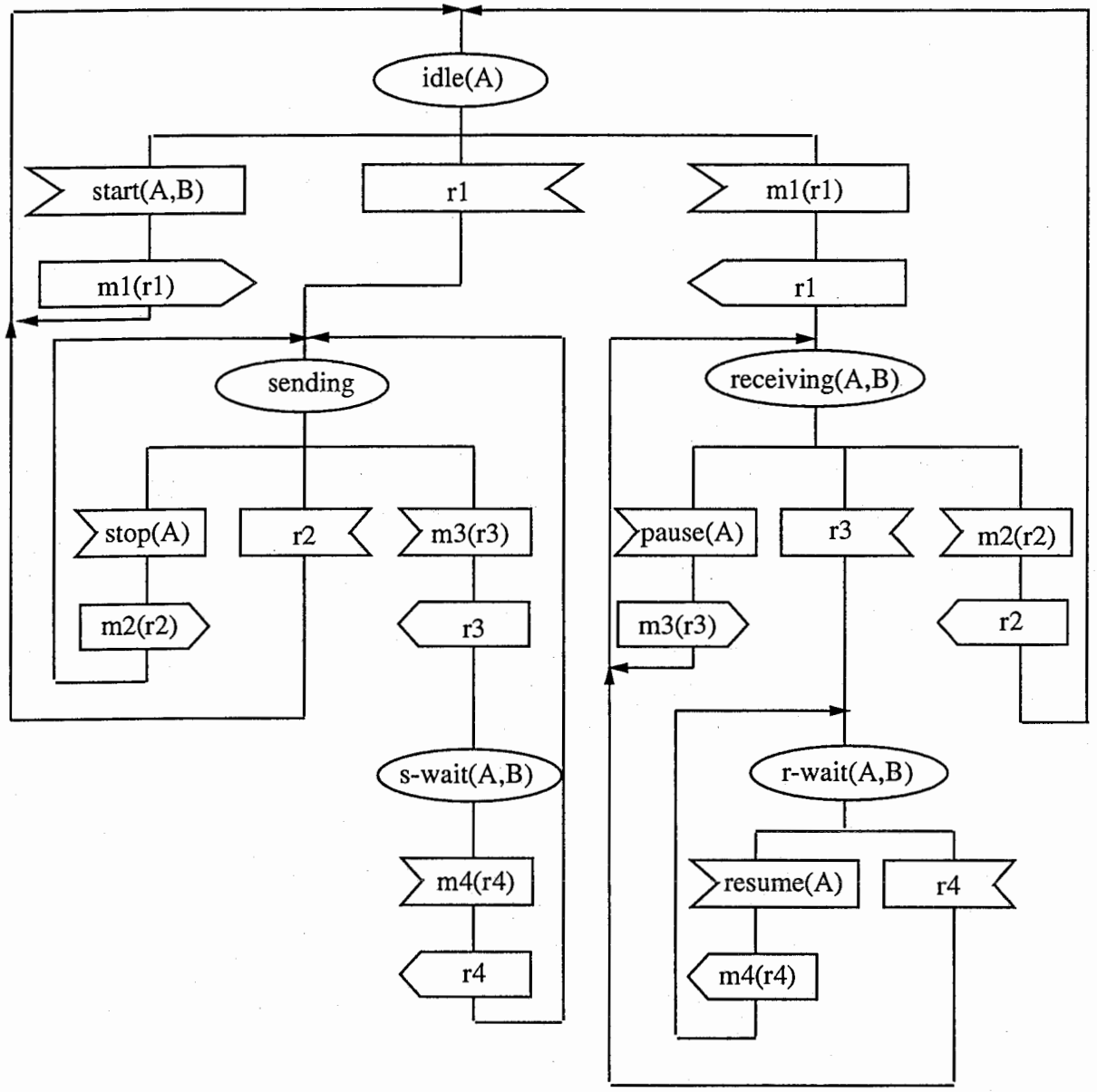


Figure 4.21: Protocol entity specification for data communication protocol.

4.6 Complexity of Communication Time

The subgraph isomorphism problem is one of NP-complete problems. We give the number of subgraphs in a graph. Let D be the maximum degree of vertices in a system graph G, d be the maximum degree of vertices in an initial graph g, and l be the depth of g. The spanning path of g is used to search for g in G. The number of graphs isomorphic to g in G are limited to the following:

$${}_DP_d({}_DP_d)^d \cdots ({}_DP_d)^{d^{l-1}} = ({}_DP_d)^{\frac{(l-1)l}{2}}$$

It follows that theoretically there will be a combinatorial explosion when searching for an initial graph to be applied in a system graph. However, in an ordinary communication service, the scope to be searched is usually not large. Therefore, the synthesized protocol usually works within a practical time. This will be shown by an experiment in Chapter 7.

Chapter 5

Protocol Synthesis for a Layered Architecture – Synthesizing Parallel Communication Protocols –

5.1 Preliminaries

Before describing a distributed algorithm for communication services written in STR, we give some definitions of graphs.

(1) $spt(g)$: A spanning tree of g. If g is a rooted graph then the root of $spt(g)$ is the same as $root(g)$.
(2) $st(v,t)$: A subtree of a tree t. The root of $st(v,t)$ is v.
(3) $sg(v,g)$: A subgraph induced from a subtree $st(v, spt(g))$.
(4) $N(v)$: A neighborhood of v. $N(v)$ is a subgraph of g such that the set of vertices of $N(v)$ is constructed from v and all adjacent vertices of v, and the set of edges of $N(v)$ is constructed from all edges incident to v.
(5) $\Gamma(v) = \{w|(v,w) \in E(spt(g))\}$.
$\Gamma(v)$ shows all children of v in $spt(g)$.

Communication time complexity
The communication time complexity is the maximum possible number of time units from start to completion of the algorithm, assuming that the inter-message delay and the propagation delay of an edge between processes are at most one time unit of some global clock. This assumption is used only for the purpose of evaluating the performance of algorithms [62].

5.2 Protocol Synthesis Algorithm

The following lemma is satisfied for subgraph isomorphism. We use this lemma to construct a distributed algorithm.

Lemma
Let $g = (V, E, v^0)$ and $g_1 = (V_1, E_1, v_1^0)$ be two rooted graphs. Let SG be a set of subgraphs of

g and SG_1 be a set of subgraphs of g_1 as follows.

$$\begin{aligned} SG &= \{N(v^0)\} \cup \{sg(v,g)|v \in \Gamma(v^0)\}, \\ SG_1 &= \{N(v_1^0)\} \cup \{sg(v,g_1)|v \in \Gamma(v_1^0)\}, \quad |SG| = |SG_1| \end{aligned}$$

If and only if the following conditions are satisfied, then g_1 is isomorphic to g.
(1) For each $sg_i \in SG$, there exists an isomorphic graph $sg_{1i} \in SG_1$.
(2) Let H be a set of isomorphic mappings in (1) such that
$H = \{h_i|h_i : V(sg_i) \longrightarrow V(sg_{1i})\}$, then the following expression is satisfied.

$$\forall v \in V,\ \forall h_i, h_j \in H,\ h_i(v) = h_j(v)$$

5.2.1 Local State

A local state has a set of adjacent process identifiers and a set of primitives representing the relationships between the adjacent processes. A local state also has a set of primitives representing a corresponding terminal state.

Let g be a graph representing a global state; then a local state in the global state is a neighborhood of a vertex of g. A local state s can be represented by a state identifier (sid) and a set of pids of adjacent processes.

$$s = (sid, P), \quad P = \{pid_0, pid_1, ...pid_k\}$$

where pid_0 shows the self identifier.

5.2.2 Message

We use the following three kinds of messages for inter-process communications.

(1) Request message

A request message is used to ask for a global state around a neighborhood process. A request message consists of a message identifier (mid). The mid of a request message req implies a set of subgraphs of the initial graphs to be checked. The set of subgraphs is denoted by $G(req)$.

(2) Response message

A response message is used for responding to a request message and consists of a mid and a set of pids. A response message res implies a set of graphs $G(res)$ and a set of pids as follows.

$$G(res) = \{g_1, ...g_k\}, \quad P(res) = \{P_1, ...P_k\}, \quad P_i = \{pid_{i0}, pid_{i1}, ...pid_{il}\}$$

where g_i is a subgraph of the initial graph. The P_i gives an isomorphic mapping h_i between g_i and a system graph as follows.

$$h_i(v_j) = pid_{ij}, \quad v_j \in V(g_i)$$

(3) Order message

An order message is used for notifying of a local state transition according to an STR rule and consists of a mid. The mid of an order message ord implies an STR rule and a vertex in the rule graph.

```
process
{
  STATE state;  MESSAGE message;

  for (;;) {
    receive(message);
    state = bpb(state, message);
  }
}
```

Figure 5.1: Distributed algorithm.

5.2.3 Algorithm Outline

A process behavior after receiving a message m at state s is called a basic process behavior and is denoted by $bpb(s, m)$. The distributed algorithm is constructed from a set of basic process behaviors as shown in Fig. 5.1.

The inter-process communications are done as follows.

$bpb(s, ev)$: When a process receives an event ev at state s, this determines a set of graphs $G(s, ev)$ whose elements may be isomorphic to subgraphs of a system graph. For each adjacent process whose pid is recorded in s, a set of graphs $G(pid, s, ev)$ is determined. Consequently, the process sends a message req to the adjacent process and the message implies $G(pid, s, ev)$.

A response message res from the adjacent process implies a set of graphs $G(res) \subset G(pid, s, ev)$. After all response messages are received, a set of graphs $G'(s, ev) \subset G(s, ev)$ is determined by analyzing the messages. Then, the process chooses a graph g in $G'(s, ev)$ and sends order messages to the related processes to achieve a global state transition according to the STR rule whose initial graph is isomorphic to g.

$bpb(s', req)$: When a process receives the message req at state s', this determines a set of graphs $G(s', req)$ whose elements may be isomorphic to subgraphs of a system graph. For each adjacent process, a set of graphs $G(pid, s', req)$ is determined. Consequently, the process sends a message req' to the adjacent process and the message implies $G(pid, s', req)$.

A response message res' from the adjacent process implies a set of graphs $G(res') \subset G(pid, s', req)$. After all response messages are received, a set of graphs $G'(s', req) \subset G(s', req)$ is determined by analyzing the messages. Then, the process returns a response message res, which implies $G'(s', req)$.

$bpb(s, ord)$: When a process receives an order message ord, it changes its state according to the STR rule implied by ord.

A basic process behavior $bpb(s, m)$ is constructed from four elements as follows.

(1) Send request messages. (2) Receive response messages.
(3) Determine a set of subgraphs. (4) Send a response message.

Figure 5.2 illustrates how an STR rule is determined by inter-process communications.

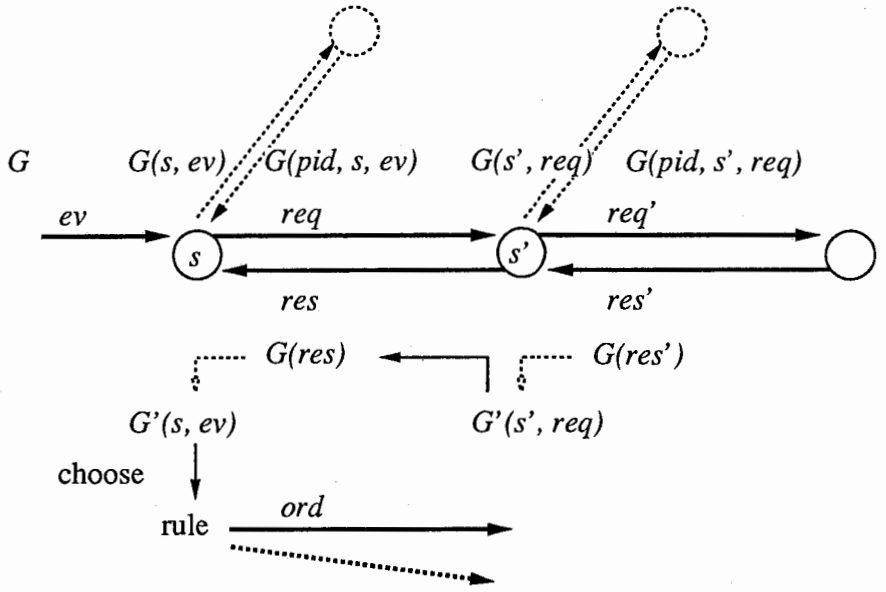


Figure 5.2: Inter-process communications.

5.3 Graph Analysis

In the distributed algorithm, a process searches for a set of subgraphs of rule graphs. To define this set of subgraphs, we analyze rule graphs.

5.3.1 Graph Resolution

A resolution tree is constructed by dividing the initial graph of an STR rule as follows.

Resolution tree
Let g and $spt(g)$ be an initial graph and a spanning tree of g. Then g is split into its subgraphs as follows.

$$g = N(v_0) \cup sg(v_1, g) \cup ...sg(v_m, g)$$

where v_0 is the root of g and v_1,...,v_m are children of v_0. Each $sg(v_i, g)$ is a subgraph of g induced from $st(v_i, spt(g))$. Each subgraph $sg(v_i, g)$ is also split into its subgraphs. A resolution tree of g called $rt(g)$ is constructed from g and $spt(g)$. There are three kinds of nodes in $rt(g)$: an event node, edge nodes, and neighborhood nodes as follows.

(1) Event node
An event node consists of the event of a rule and has one child. The event node is the root of a resolution tree, and the child of the event node is a neighborhood node corresponding to the root of g.

(2) Edge node
An edge node edv corresponds to an edge of $spt(g)$ and is constructed from two kinds of elements as follows.

$$edv = (LABEL(edv), sg(edv))$$

$LABEL(edv)$ is a set of labels attached to the edge. $sg(edv)$ is an induced subgraph $sg(v', g)$ when v' is the terminal vertex of the edge in g. The children of the edge node are neighborhood nodes.

(3) Neighborhood node
A neighborhood node nv corresponds to a vertex v of g and is constructed from three kinds of elements as follows. The children of the neighborhood node are edge nodes.

$$nv = (N(nv), sg(nv), F(nv))$$

where $N(nv)$=(V_0, E_0) is a neighborhood of $v(N(v))$, $sg(nv)$=(V, E) is an induced subgraph of $g(sg(v, g))$, and $F(nv)$ is a set of mappings on V as follows.

Mappings on a set of vertices

$$\begin{aligned} &F(nv) = \{f(nv), f(edv_1), ...f(edv_k)\} \\ &f(edv_i) : V \longrightarrow (V_i \cup \{\lambda\}) \\ &f(edv_i)(v) = \begin{array}{ll} v & (\text{if } v \in V_i), \\ \lambda & (\text{otherwise}) \end{array} \end{aligned}$$

where edv_i is a child node of nv. The graph $sg(edv_i) = (V_i, E_i)$ contained in edv_i is a subgraph of $sg(v, g)$.

Let R be a set of STR rules and G be a set of initial graphs of R. For each graph $g \in G$, we construct a resolution tree $rt(g)$. The set of resolution trees for the graphs in G is denoted by $RT(G)$. The spanning trees used for generating resolution trees of $RT(G)$ satisfy the following condition.

$$\forall g_i, g_j \in G, \ \ (g_i \sqsubset g_j \Longrightarrow spt(g_i) \sqsubset spt(g_j))$$

5.3.2 Graph Synthesis

A synthesis tree $SYT(G)$ is constructed by synthesizing the resolution trees in $RT(G)$ into one tree.

Synthesis Tree

A synthesis tree consists of four kinds of nodes as follows.

(1) Root node
A root node has no internal structure. The children of a root node are event nodes.

(2) Event node
An event node ev consists of two elements. One is an event denoted by $event(ev)$ and the other is a set of graphs denoted by $G(ev)$. The children of an event node are neighborhood nodes.

$$ev = (event(ev), G(ev))$$

(3) Edge node
An edge node edv consists of two elements. One is a set of labels $LABEL(edv)$ and the other is a set of graphs denoted by $G(edv)$. The children of an edge node are neighborhood nodes.

$$edv = (LABEL(edv), G(edv))$$

(4) Neighborhood node

A neighborhood node nv consists of two elements. One is a neighborhood $N(nv)$ and the other is a set of graphs denoted by $G(nv)$. For each graph in $G(nv)$, a set of mappings is attached. These mappings are similar to the nodes in a resolution tree. The structure of nv is described as follows.

$$\begin{aligned} nv &= (N(nv), G(nv)) \\ G(nv) &= \{SG_1(nv), ...SG_k(nv)\} \\ SG_i(nv) &= (sg_i(nv), F_i(nv)) \end{aligned}$$

For each rooted graph $sg_i(nv) = (V_i, E_i)$, the neighborhood of the root is isomorphic to the neighborhood $N(nv) = (V_{i0}, E_{i0})$.

For each child node of nv, there is a corresponding mapping in $F_i(nv)$.

$$F_i(nv) = \{f_i(nv), f_i(edv_1), ...f_i(edv_k)\}$$

where edv_j is a child node of nv.

In the set of graphs $G(edv_j)$, there is a graph $sg_{ij} = (V_{ij}, E_{ij})$ isomorphic to a subgraph of $sg_i(nv)$. The $f_i(edv_j)$ is a mapping as follows.

$$\begin{aligned} &f_i(edv_j) : V_i \longrightarrow (V_{ij} \cup \{\lambda\}) \\ &f_i(edv_j)(v_i) = \begin{cases} v_{ij} & \text{(if there is a corresponding vertex in } V_{ij}\text{)}, \\ \lambda & \text{(otherwise)} \end{cases} \end{aligned}$$

The graph sg_{ij} is determined by $sg_i(nv)$ and edv_j, so it is denoted by $sg(sg_i(nv), edv_j)$. Figure 5.3 illustrates a synthesis tree.

Synthesis Tree Construction

Let G be a set of initial graphs of STR rules, and $G_k = \{g_1, ...g_k\}$ be a subset of G. Then, synthesis tree $SYT(G)$ is constructed recursively as follows.

(I) $SYT(G_1)$

Synthesis tree $SYT(G_1)$ is constructed by creating a root and an edge that connects the root with $rt(g_1)$.

(II) $SYT(G_{k+1})$

$SYT(G_{k+1})$ is constructed from $SYT(G_k)$ and $rt(g_{k+1})$ as follows, where $G_{k+1} = G_k \cup \{g_{k+1}\}$.

Let ev, nv, edv be an event node, a neighborhood node, and an edge node of $SYT(G_k)$. Let ev_{k+1}, nv_{k+1}, edv_{k+1} be an event node, a neighborhood node, and an edge node of $rt(g_{k+1})$.

(1) Event node synthesis

Compare events in ev_{k+1} and ev.

If ev and ev_{k+1} have the same event, then add g_{k+1} to $G(ev)$, and for each $nv_{k+1} \in \Gamma(ev_{k+1})$ and $nv \in \Gamma(ev)$, synthesize $st(nv_{k+1}, rt(g_{k+1}))$ with $st(nv, SYT(G_k))$ as described below in (2).

If any event node in $EVENT(G_k)$ does not have the same event as ev_{k+1} has, then add $rt(g_{k+1})$ to $root(G_k)$.

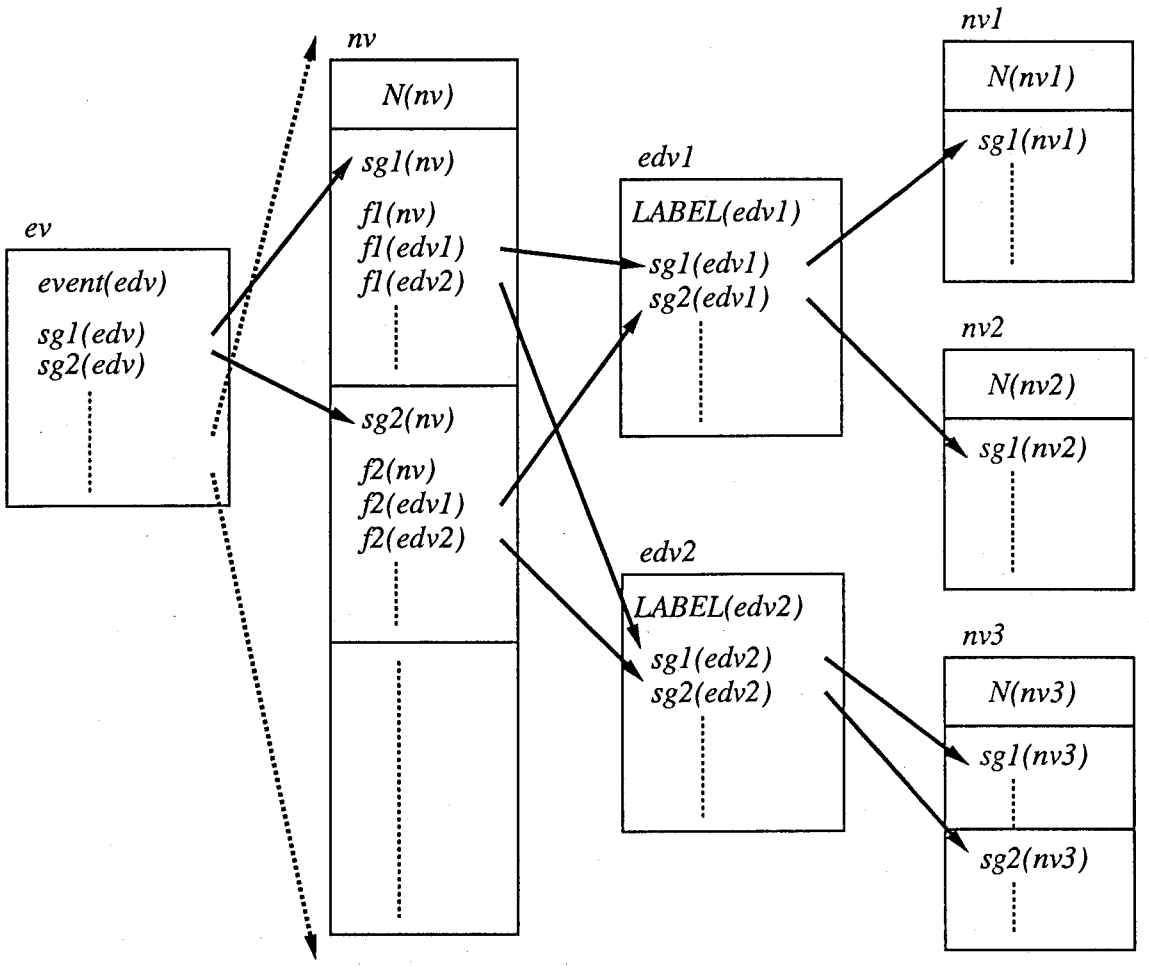


Figure 5.3: Synthesis tree.

(2) Neighborhood node synthesis
Compare neighborhoods in nv_{k+1} and nv.

If $N(nv)$ is identical to $N(nv_{k+1})$, then synthesize nv_{k+1} with nv as described below in (2-1).

If any node in $\Gamma(ev)$ does not have $N(nv_{k+1})$, then add $st(nv_{k+1}, rt(g_{k+1}))$ to ev.

(2-1) G(nv) synthesis
Let $sg(nv)$ be an element of $G(nv)$. Compare $sg(nv_{k+1})$ and $sg(nv)$.

If $sg(nv)$ is isomorphic to $sg(nv_{k+1})$, then $st(nv_{k+1}, rt(g_{k+1}))$ is already included in $st(nv, SYT(G_k$. Therefore, $st(nv, SYT(G_k))$ is not modified.

If any graph in $G(nv)$ is not isomorphic to $sg(nv_{k+1})$, then add $sg(nv_{k+1})$ to $G(nv)$.

For each $edv_{k+1} \in \Gamma(nv_{k+1})$, there is a node $edv \in \Gamma(nv)$ such that $LABEL(edv) = LABEL(edv_{k+1})$ because $N(nv)$ is identical to $N(nv_{k+1})$. Then synthesize $st(edv_{k+1}, rt(g_{k+1}))$ with $st(edv, SYT(G_k))$ as described below in (3).

(3) Edge node synthesis
Let $sg(edv)$ be an element of $G(edv)$. Compare $sg(edv_{k+1})$ and $sg(edv)$.

If $sg(edv)$ is isomorphic to $sg(edv_{k+1})$, then $st(edv_{k+1}, rt(g_{k+1}))$ is already included in $st(edv, SYT(G_k))$. Therefore, $st(edv, SYT(G_k))$ is not modified.

If any graph in $G(edv)$ is not isomorphic to $sg(edv_{k+1})$, then add $sg(edv_{k+1})$ to $G(edv)$.

For each $nv_{k+1} \in \Gamma(edv_{k+1})$ and $nv \in \Gamma(edv)$, synthesize subgraphs as described in (2).

5.4 Distributed Algorithm Generation

To generate a distributed algorithm, we first generate a set of local states and messages. We then generate a set of basic process behaviors.

In this paper, we assume that the number of labels a process may have is restricted; accordingly there is a maximum number of labels (maxlabel).

5.4.1 State Generation

Let s, r, v be a local state, an STR rule, and a vertex of the rule graph of r. Let $N(r, v)$ and $N'(r, v)$ be a neighborhood of v in the initial graph and the next graph of r.

If $N(r, v) \sqsubset s$, then let s' be a local state that is generated from s by changing $N(r, v)$ to $N'(r, v)$. If the number of labels of s' does not exceed maxlabel, then the STR rule may be applied to s. The next state is denoted by $next(s, r, v)$.

The set of local states S is generated recursively as follows.

$$(I) \quad S_0 = \{N(r, v) | r \in R, v \in V(r)\}$$
$$(II) \quad S_{i+1} = S_i \cup \{next(s, r, v) | s \in S_i, r \in R, v \in V(r)\}$$

where $V(r)$ is a set of vertices in the rule graphs of r. S becomes a finite set.

5.4.2 Synthesis Tree Modification

Let $node$ and $S(node)$ be an event or edge node of $SYT(G)$ and a set of neighborhoods in the node in $\Gamma(node)$. Let s be a local state in S, and let $S(node, s)$ and $NV(node, s)$ be a subset

of $S(node)$ and a subset of $\Gamma(node)$ as follows.

$$\begin{aligned} S(node, s) &= \{state \in S(node) |\ state \sqsubset s\} \\ NV(node, s) &= \{nv \in \Gamma(node) |\ N(nv) \in S(node, s)\} \end{aligned}$$

(1) If $S(node)$ contains s, there is a node $nv(s) \in \Gamma(node)$ that satisfies $N(nv(s)) = s$.

If $S(node)$ does not contain s, then create a new neighborhood node $nv(s)$ as a child of the *node* as follows.

$$N(nv(s)) = s, \quad G(nv(s)) = \phi$$

(2) For each node nv in $NV(node, s)$, add $G(nv)$ to $G(nv(s))$ in order. Then synthesize a subtree $st(nv, SYT(G))$ to $st(nv(s), SYT(G))$ as described earlier ($G(nv)$ synthesis).

5.4.3 Message Generation

For each vertex of a rule graph, an order message is generated. A set of request messages and response messages are generated from synthesis tree $SYT(G)$.

For each edge node edv, a request message $req(edv)$ is generated. The $req(edv)$ implies a set of graphs $G(edv)$.

For each neighborhood node nv, create a set of subsets $GSET(nv)$ as follows.

$$GSET(nv) = \{G_i | G_i \subset G(nv),\ (\forall g_j, g_k \in G_i, \neg(g_j \sqsubset g_k))\}$$

For each element $G_i \in GSET(nv)$, a response message res(nv, G_i) is generated.

5.4.4 Process Behavior Generation

Let *node* be an event or edge node in $SYT(G)$ that corresponds to message m. Let nv be the child of the *node* whose neighborhood is identical to state s. A basic process behavior $bpb(s, m)$ is generated as follows.

(1) Send request messages

Let edv_i be a child node of nv. A request message to the adjacent process $req(edv_i)$ is generated from edv_i. The destination of $req(edv_i)$ is the process connected with the labels of edv_i.

(2) Receive response messages

A set of response messages for $req(edv_i)$ is generated from the child nodes of edv_i. Let $res(edv_i)$ be a response message for $req(edv_i)$. Then the following is satisfied.

$$G(res(edv_i)) \in \bigcup_{nv_j \in \Gamma(edv_i)} GSET(nv_j)$$

(3) Determine a set of isomorphic subgraphs

(3-1) Response message id comparison

For each $edv_i \in \Gamma(nv)$ and for each $sg_j = (V_j, E_j) \in G(nv)$, confirm whether the implied set of graphs $G(res(edv_i))$ contains a subgraph $sg(sg_j(nv), edv_i) = (V_{ji}, E_{ji})$. Then create a set of graphs G_1 as follows.

$$G_1 = \{sg_j \in G(nv) | \forall edv_i \in \Gamma(nv),\ sg(sg_j(nv), edv_i) \in G(res(edv_i))\}$$

(3-2) Pid comparison

Let P_{ji} be a set of pids in $res(edv_i)$ that corresponds to $sg(sg_j(nv), edv_i)$. P_{ji} gives a mapping $h_{ji} : V_{ji} \longrightarrow PIDS$ where $PIDS$ is the set of process identifiers in a system.

The node nv contains a mapping $f_j(edv_i) : V_j \longrightarrow V_{ji}$. We define the mapping $\bar{h_{ji}}$ as follows.

$$\begin{array}{ll} \bar{h_{ji}} : V_j \longrightarrow (PIDS \cup \{\lambda\}) & \\ \bar{h_{ji}}(v) = \begin{array}{ll} h_{ji} \cdot f_j(edv_i)(v) & (\text{if } f_j(edv_i)(v) \neq \lambda), \\ \lambda & (\text{otherwise}) \end{array} & \end{array}$$

For each $sg_j \in G_1$, for each $v \in V_j$, and for each $edv_i, edv_k \in \Gamma(nv)$, confirm whether $\bar{h_{ji}}(v)$ is identical to $\bar{h_{jk}}(v)$. Then create a set of graphs G_2 as follows.

$$G_2 = \{sg_j \in G_1 | \forall v \in V_j,\ \forall edv_i, edv_k \in \Gamma(nv),\ \bar{h_{ji}}(v) = \bar{h_{jk}}(v)\}$$

(4) Send a response message

We can define the mapping $h_j : V_j \longrightarrow PIDS$ as follows. For each vertex $v \in V_j = V(sg_j)$, there is a mapping $\bar{h_{ji}}$ and $\bar{h_{ji}}(v) \neq \lambda$. Then $h_j(v)$ is defined as $\bar{h_{ji}}(v)$, which is a pid in $PIDS$.

Create a response message $res(nv)$ as follows.

$$\begin{array}{l} G(res(nv)) = G_2, \quad P(res(nv)) = \{P_1, P_2, ...P_k\}, \\ P_j = \{h_j(v) | v \in V_j,\ sg_j \in G_2\} \end{array}$$

5.5 Evaluation

Let d be the depth of $SYT(G)$. It takes at most d time units to send request messages. It also takes at most d time units to send response messages. It takes one time unit to send order messages. Therefore, the communication time complexity of the distributed algorithm becomes 2d+1.

The depth of $SYT(G)$ is identical to the depth of the deepest resolution tree in $RT(G)$. Therefore, the communication time complexity does not depend on the number of STR rules; it only depends on the STR rule that has the largest number of processes.

The computation complexity of each process is determined by the efficiency of the response message id comparison and the pid comparison. The former efficiency is proportional to $|\Gamma(nv)||G(nv)|^2$, and the latter efficiency is proportional to $|\Gamma(nv)||V_j||G(nv)|$.

Let n be the number of STR rules. $|\Gamma(nv)|$ and $|V_j|$ are considered constants for n. When estimating the number of graphs in a neighborhood node $|G(nv)|$ proportional to $log(\mathrm{n})$, the computation complexity becomes $O((log(\mathrm{n}))^2)$.

Chapter 6

Software Specification Generation from Protocol Specifications

In the previous two chapters we showed protocol synthesis algorithms. In this chapter, we define a detailed specification description language STR/D (Detailed Specification Language for STR).

6.1 Detailed Specification Language STR/D

STR/D describes supplementary specifications to implement service specifications described with STR. An STR/D specification consists of a set of STR/D rules. An STR/D rule describes tasks to be executed on the state transitions of terminals by STR rules. Each STR/D rule has the syntax:

position-designation {task-designation}

This rule specifies that "task-designation" is executed at positions where the condition "position-designation" is satisfied on a state transition of a protocol entity.

6.1.1 Position Designation

A synthesized protocol entity specification consists of local states, inputs and other elements. To designate positions in a protocol entity specification we use local states and message inputs. The local states and message inputs depend on the state primitives and events of STR rules. This implies that we can describe STR/D rules by knowing state primitives and inputs that are being used or are to be used in STR rules. If we describe STR/D rules with state primitives and events but without depending on individual protocol entity specifications, the STR/D rules turn out to define meanings of these STR elements. If new state primitives and events are used for describing a service, it is necessary to define new STR/D rules to specify them as a matter of course.

Positions are designated by states, inputs and their combinations.

State designation

Local states in a protocol entity are described by a set of state primitives, so we use state primitives for designating local states. States are specified by *initial*, *terminal*, *state*, *next-state*, *primitive* and *next-primitive*.

initial A rule for the initial position designates initial tasks to begin services. Position 1 in Fig. 6.1 is designated by *initial.*

terminal A rule for the terminal position designates final tasks before returning to the initial state. Position 6 in Fig. 6.1 is designated by *terminal.*

state State designates the positions just after inputs at the specified state. Positions 2, 3 and 4 in Fig. 6.1 are designated by *state.*

next-state Next-state designates the position just before the specified state. Position 5 in Fig. 6.1 is designated by *next-state.*

primitive Primitive designates the positions just after inputs at the states that have specified state primitives. Positions 2, 3 and 4 in Fig. 6.1 are designated by *primitive.* The difference between *state* and *primitive* is whether a state is completely specified or partially-specified.

next-primitive Next-primitive designates the positions just before the states that have specified state primitives. Position 5 in Fig. 6.1 is designated by *next-primitive.*

Among these state designations, *state* and *primitive* can be combined with *next-state* and *next-primitive.* Such a combination designates the position 5 in Fig. 6.1.

Input designation

Inputs designate the positions just after receiving the designated message. There are four types of inputs: *event*, *request*, *respond* and *norule.* Every position after receiving a message of the specified type is designated if only an input type is specified. As for *event* we can specify individual event names. Input designation can be combined with state designations. Positions 2, 3 and 4 in Fig. 6.1 are designated by *input.*

State transition designation

State transition paths are designated by describing the difference in primitives in the current and the next states. The syntax is as follows.

transition (<state transition expression>)

The state transition expression in this rule is described as the following syntax (1) or (2).

(1) + (a set of state primitives)
(2) − (a set of state primitives)

Syntax (1) describes the state primitives not included in the current states but in the next states, and syntax (2) describes the state primitives not included in the next states but in the current states. Position 5 in Fig. 6.1 is designated by *transition.*

6.1.2 Task Designation

Tasks described in "task-designation" conform to C statements. These tasks are separated by semicolons if multiple tasks are specified in a single task-designation. We can describe a conditional statement in a task-designation. This statement will be used to change the next state according to the status of a task. A conditional statement is written by the following syntax.

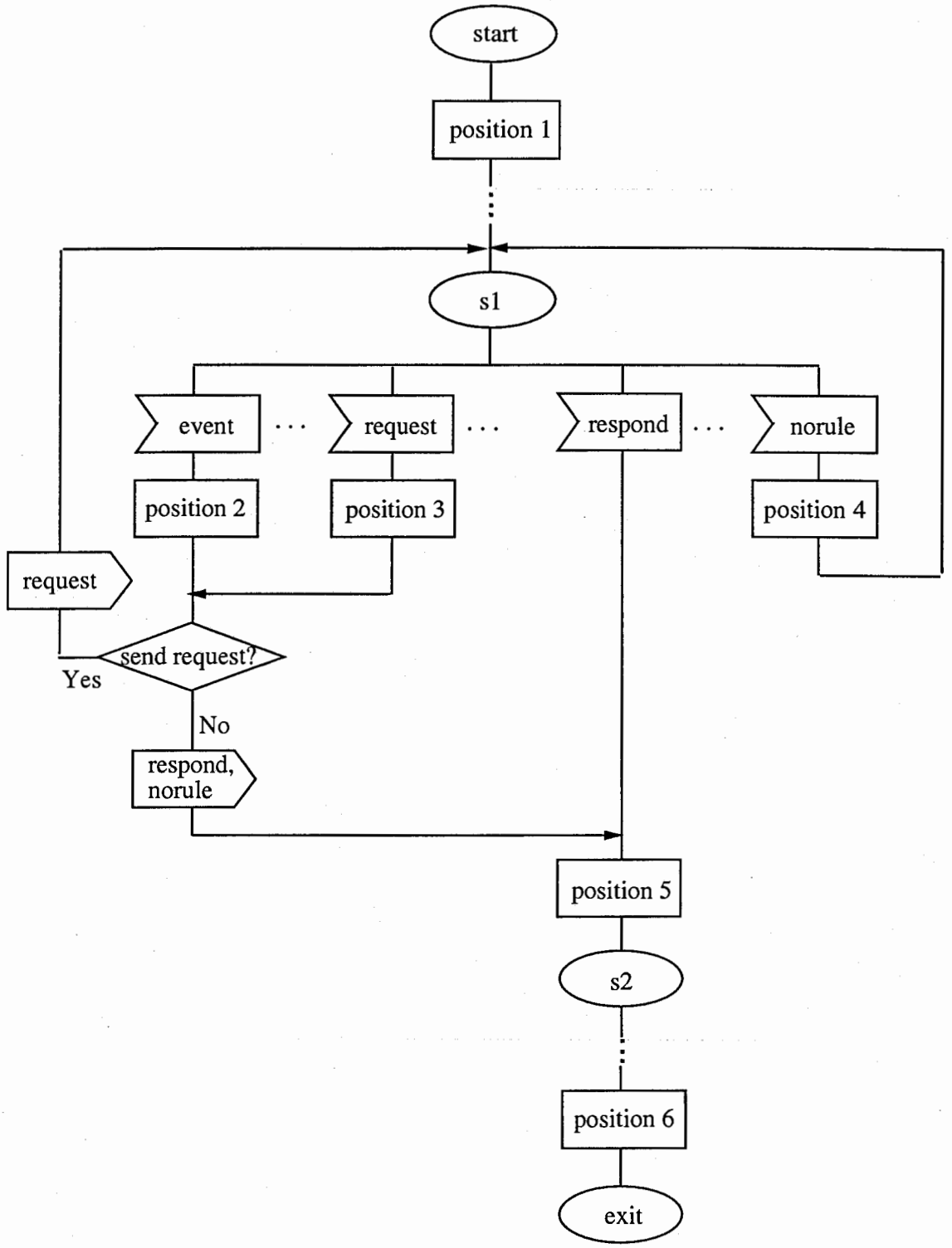


Figure 6.1: Task insertion.

```
primitive(dial-tone(A)) {tone_on(DLTONE);}                          (r1)
state_input((dial-tone(A)) & (event dial(A,B))) {tone_on(RBTONE);}  (r2)
transition(-(idle(A)) +(ringing(A,B))) {tone_on(RGTONE);}           (r3)
```

Figure 6.2: Example of STR/D rules.

unless(< C expression >) {< C statements >}
< state designation >;

This statement describes a specification when a condition is not satisfied. If the C expression is not satisfied, the C statements in this statement are executed and the protocol entity changes its state to the designated state.

6.2 Example

Figure 6.2 shows an example of an STR/D description corresponding to Fig. 1.3. Rule r1 indicates that tone_on(DLTONE) is executed just before entering a state having the primitive "dial-tone(A)". Rule r2 indicates that tone_on(RBTONE) is executed when the event dial(A,B) occurs at a state that contains the primitive dial-tone(A). Rule r3 indicates that tone_on(RGTONE) is executed on a transition where the primitive "idle(A)" is deleted and the primitive "ringing(A,B)" is added at the next state.

The following example gives the STR/D rules for implementing pots in Fig. 1.4 on a PBX.

```
var     A, B, C, U;
start {Initialize();}
term {Stop();}
transition ( -(dial-tone(A)) +(ring-back(A,B)) )
               { Stop_dial(); Tone_off(DT); R_path(B); Tone_on(RBT); }
transition ( -(ring-back(A,B)) +(path(A,B)) ) { Tone_off(RBT); Connected(B); }
transition ( -(path(A,B)) +(idle(A)) ) { F_path(B); Disconnect(B); }
transition ( -(idle(A)) +(ringing(A,B)) ) { Tone_on(RNG); }
transition ( -(ringing(A,B)) +(path(A,B)) ) { Tone_off(RNG); Connect(B); }
transition ( -(path(A,B)) +(busy(A)) )
               { Disconnected(B); Tone_on(BT); }
transition ( -(busy(A)) +(idle(A)) ) { Tone_off(BT); }
transition ( -(ring-back(A,B)) +(idle(A)) ) { F_path(B); Tone_off(RBT); }
transition ( -(ringing(A,B)) +(idle(A)) ) { Tone_off(RNG); }
transition ( -(dial-tone(A)) +(busy(A)) )
               { Stop_dial(); Tone_off(DT); Tone_on(BT); }
transition ( -(idle(A)) +(dial-tone(A)) ) { Tone_on(DT); Start_dial(NML); }
transition ( -(busy(A)) +(idle(A)) ) { Tone_off(SBT); }
transition ( -(dial-tone(A)) +(idle(A)) ) { Stop_dial(); Tone_off(DT); }
```

We note that every rule is described by state transition designations in this example.

Chapter 7

Application to PBX Software Generation

This chapter shows results of an automated software generation method applied to PBX systems. First we show the results of a specification description task to implement seven typical services on a PBX, and then clarify the effectiveness of the proposed software generation method. We then describe results obtained by implementing one of these services on two kinds of PBX systems and again evaluate the proposed method. Software is obtained as follows. First, protocol specifications are synthesized from service specifications using the algorithm in Chapter 4 and then the protocol specifications are transformed into software specifications by the refinement method in Chapter 6.

7.1 Results of Specification Description

Communication software is usually developed incrementally to provide new services. Services such as POTS, CCBS, CW, CFV, TWC, UPT and TCS have been implemented according to this development style. These services are defined as follows.

POTS The Plain Old Telephone Service (POTS) involves an ordinary call between two stations.

CCBS A customer subscribing to the Completion of Call to Busy Subscriber (CCBS) service can, after reaching a busy station, hang up first, and then dial the activation code to activate CCBS. When the busy station goes on-hook and the calling station is on-hook, the calling station is rung. Upon an answer, the call is automatically completed to the station previously busy.

CW The Call Waiting (CW) service informs a busy station user that another call is waiting. The busy station may answer the new call by one of two methods. One is with a flash, placing the original call on hold and answering the new call. The other is for the busy station user to go on-hook, in which case the station user is rung and connected to the new call upon an answer.

CFV The Call Forwarding Variable (CFV) service allows a station to redirect calls intended for that station (base station) to another station (remote station). A subscriber can activate the service in two ways. The first way is as follows: The subscriber goes off-hook, receives

a dial tone, and dials the service prefix followed by a 2-digit CFV activation code or dials the 2-digit CFV activation code at the end of the dialing signal. A recall dial tone is then heard by the subscriber. At this time, the Directory Number of the remote station is dialed and the system attempts to complete the call in the normal manner. If the remote station answers, CFV is activated. The second way to activate the CFV service can be used if the remote station does not answer the call or is busy. The subscriber merely repeats the same procedure as described in the previous text within 2 minutes of hanging up from the first attempt. On this second activation the system applies a confirmation tone following a delay of at least 1 second of a dial tone to the base station, rather than attempting to complete the call to the remote station. A receipt of confirmation tone tells the subscriber that CFV has been activated. If the timed interval has elapsed, a second request for service activation is processed as an initial request.

If the subscriber has the call forwarding feature activated and receives a call, a ring reminder is applied to the base station (when idle), to indicate that the call has been received and forwarded. The subscriber cannot answer calls at the base station while CFV is active, but can originate calls.

To deactivate call forwarding, the subscriber goes off-hook, receives a dial tone, and dials a deactivation code; a confirmation tone is then returned to the customer. Dial tone returns after the confirmation tone.

TWC Three-Way Calling (TWC) is a custom calling feature that allows a customer to add a third party to an existing conversation without operator assistance. The party initiating TWC may put one party on hold in private while dialing or talking to another party and can later include the party on hold. The added party may be dropped from the connection with a flash from the initiating party.

UPT Universal Personal Telecommunication (UPT) enables access to telecommunication services while allowing personal mobility. It enables each UPT User to participate in a user-defined set of subscribed services and to initiate and receive calls on the basis of a personal, network-transparent UPT Number across multiple networks at any terminal, fixed or mobile, irrespective of geographic location, limited only by terminal and network capabilities and restrictions imposed by the network operator.

TCS Terminating Call Screening (TCS) screens incoming calls against a screening list consisting of time and originating terminal directory numbers. In this experiment, the screened call activates either POTS, message transfer or call transfer.

Seven services S1, $\cdots$, S7 are defined as a combination of these services.

S1 POTS itself.

S2 A specification consisting of S1 and CBS.

S3 A specification consisting of S2 and CW.

S4 A specification consisting of S3 and CFV.

S5 A specification consisting of S4 and TWC.

S6 A specification consisting of S5 and UPT.

Table 7.1: Number of rules and primitives in library software

Service	STR rules	STR/D rules	Library
S1	23	14	10
S2	46 (+23)	16 (+2)	10 (+0)
S3	80 (+34)	20 (+4)	12 (+2)
S4	129 (+49)	26 (+6)	12 (+0)
S5	207 (+78)	26 (+0)	12 (+0)
S6	348 (+141)	72 (+46)	16 (+4)
S7	379 (+31)	76 (+4)	16 (+0)

Table 7.2: Ratio of added rules and primitives in library software

Added service	STR rule	STR/D rule	Library
CCBS (S1 → S2)	50	13	0
CW (S2 → S3)	43	20	17
CFV (S3 → S4)	38	23	0
TWC (S4 → S5)	38	0	0
UPT (S5 → S6)	41	64	25
TCS (S6 → S7)	8	5	0
Average	36	29	7

S7 A specification consisting of S6 and TCS.

Table 7.1 shows the results of the experiment. Each figure shows the number of rules or primitives in the library software, and each figure in parentheses shows the number of supplementary parts compared with that of the previous service.

The library software was developed manually, and is of a small size. In fact, the total size of the library software is about two thousand lines in the C language. The library software was developed by using primitive software provided by a platform beforehand. Table 7.2 lists the percentages of added parts in Table 7.1. Table 7.3 represents optimization ratios in communications among processes.

7.2 Evaluation of Description Results

Table 7.2 shows that the average ratios of added STR rules, STR/D rules and library software are 36%, 21% and 7%, respectively. A more detailed knowledge of communications software and systems is necessary to do development in the order of library software, STR/D rules and STR rules. Supplementary library software is expected to be unnecessary or very small in size after services have been developed to a certain extent.

Furthermore, the following semi-automated addition will be expected as far as STR/D rules are concerned. STR/D rules can possibly be generated semi-automatically by defining the semantics of primitives in STR rules with a conceptual model of communication services.

Table 7.3: Ratio of the number of times and time in communications

Service	Times (%)	Time (%)
S1	57	57
S2	53	49
S3	56	51
S4	43	37
S5	42	34
S6	36	28
S7	33	25
Average	46	40

The proposed method can be applied to cases where non-experts are developing communication services. In this case the software development described above will make it possible to implement the services on a communications system. Consequently, almost all of the parts can be automatically or semi-automatically generated by the non-experts in software development. These facts imply that the proposed automated software generation method can be applied to communication service development by non-experts.

We evaluated the effectiveness of the method in an application. A PBX of a large size with 3,600 lines maximum was used in this experiment. About 30% of the software was dedicated to service control in the PBX.

Next we evaluated the optimization method described in this thesis. Table 7.3 shows ratios of optimization in communications for an application of appropriate rules in the algorithm of Chapter 4 compared with a naive algorithm. The optimization ratios become big in large services. In the supposed development style of communications software by non-experts, it is assumed that services have already been equipped to a certain extent. It therefore follows that communications is expected to be below 33% and 25% in the number of times and time, respectively.

7.3 Software Architecture

We show a software architecture for controlling communication services using a generated program. This architecture is developed on two PBX systems. It aims to minimize the target hardware dependent parts and to maximize the customizability of the application interface used by STR and STR/D rules. This software architecture can be easily applied to a distributed system consisting of multiple nodes.

7.3.1 Processes

There are five kinds of processes: STRP, INP, OUTP, SMP and TIMER as shown in Fig. 7.1. In the following, we explain these processes and their roles.

STRP processes are automatically generated from the STR and STR/D rules. An STRP process is created when a call is originated or terminated at a terminal in the idle state.

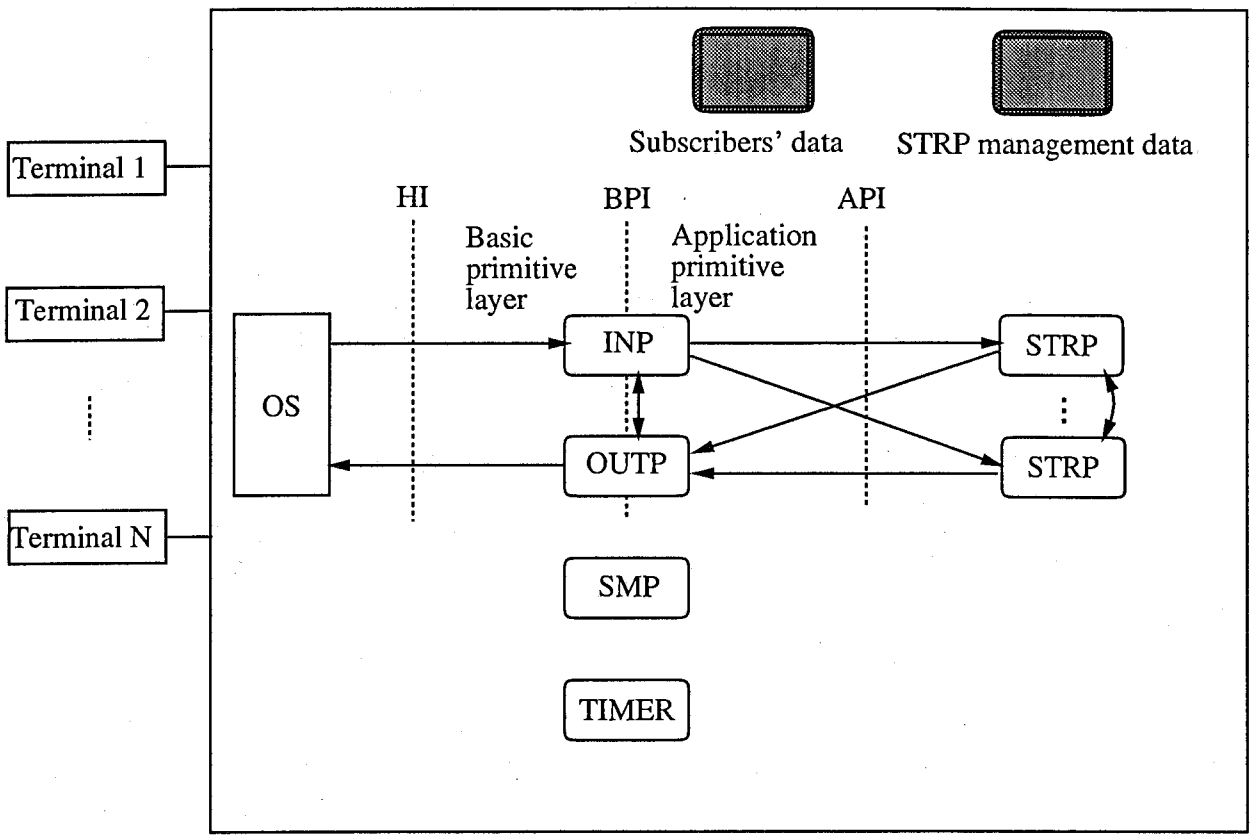


Figure 7.1: System architecture.

INP converts inputs from a terminal to events which are specified in STR rules. INP has converting and buffering functions for inputs from terminals through the OS (Operating System), and also has a function for creating STRP processes. An event may be made from a sequence of inputs and sometimes multiple events are made from one input from a terminal. Since all inputs from terminals go through the INP, the INP can provide a function for mutual exclusion of rule application with the semaphore.

OUTP converts outputs from STRP to PBX control primitives which can control a PBX hardware unit directly. OUTP has converting and buffering functions for outputs to the OS.

SMP records events and communications between processes for logging.

TIMER is used as a timer.

INP and OUTP have two layers, the basic primitive layer for customizability of the application interface and the application primitive layer for the logical interface of STRP processes.

7.3.2 Logical Interface

There are two layers of PBX control primitives: basic primitive and application primitive layers. They are implemented within INP and OUTP. The interface between OS and INP, or OUTP is called the hardware interface (HI). The interface between basic primitives and application primitives is called the basic primitive interface (BPI), and the interface between application primitives and STRPs is called the application primitives interface (API). Application primitives are made from basic primitives. BPI is independent of any specific hardware. Consequently, we can customize API by using BPI without hardware knowledge. The major difference between API and BPI is terminal identification. In BPI a terminal is expressed by a physical address, while in API a terminal is identified by a logical number, i.e., a telephone number.

Basic primitive interface

The basic primitive interface is established in order to add interfaces to new events and new tasks without using hardware control primitives. Events and tasks may increase in number when new services are developed by using new events or new state primitives.

- Each input from a hardware or a timer basically corresponds to one event. A sequence of dials is valid only if it identifies a terminal. In STR an event is defined as a logical input from a terminal rather than an actual input. In other words an event is set to the largest input sequence which does not have to be divided when a new service is added.

- Primitives for controlling a hardware unit are set so that one primitive is used for one objective. When we use these primitives, it is not necessary to designate a means for achieving the objective. For example, it is not necessary to designate a speech path or a conference trunk to connect terminals. This enables one to connect terminals without knowing the other party's situation.

Application primitive interface

API is the interface for events and tasks used in STR and STR/D. Basically we do not designate hardware resources unless we need them for services, such as resource reservation.

Table 7.4: Execution time.

	Execution time	C dynamic steps
STRP/MMP	1.1	1.2

Table 7.5: Number of states and transitions.

	STRP	MMP	STRP*
Number of states	12	9	11
Number of transitions	63	26	41

Application primitives are made from basic primitives. We have described such things as pots, three way calling service, call forwarding service, call waiting service, call completion service for a busy subscriber, and Universal Personal Telecommunication, with the current API which does not designate hardware resources. Each STRP is assigned to a terminal by INP for an origination and by communication primitive "send" for receiving a terminating call.

7.4 Implementation Results

We show the execution time for an individual service dependent part in the developed PBX. We compare STRP and MMP which is an optimized program.

A simplified POTS is used to compare the execution times of STRP and MMP. STRP and MMP have the same architecture. Each terminal is controlled by a process, and each process communicates with other processes to know its state. MMP is a dedicated program to implement pots. Table 7.4 shows the result. "Execution time" represents the consumed time of the processes, and "C dynamic steps" shows the steps of the C source program. According to the result the automatically generated program STRP runs in comparable time with MMP.

The execution time depends on the transition times and message size from call origination to call termination. Table 7.5 shows the size of state transitions. STRP is obtained from 15 STR rules and 16 STR/D rules. STRP communicates two times, i.e., to examine the state of the other STRP and to know the rule to be applied. On the other hand, MMP does not need to know the state of the other MMP via communication after the first communication, because MMP does not change its local state on account of a call from a third party. Therefore, MMP communicates less times using less messages than STRP. STRP, unlike MMP, sends a message which has its local state, i.e., STRP communicates with MMP for synchronization. Further work is needed for optimization. STRP* is a manually optimized STRP.

Chapter 8

Software Generation for Functional Model

The protocol synthesis methods in Chapters 4 and 5 produce a protocol assuming a layered architecture model. There is another protocol architecture called the functional model [7]. Universal Personal Telecommunication [44] has been standardized in order to be provided using the functional model in Fig. 1.7. In the functional model functions are not layered but distributed. These functions are distributed in functional entities. The software generation method in this chapter can be applied to this functional model.

8.1 Stepwise Refinement

We show a method of generating software conforming to any functional model from STR and STR/D rules. First we obtain the "service specifications" of each functional entity from STR rules which specify terminal behaviors, and STR/D rules. Then we synthesize finite state machine based functional entity specifications from the obtained service specifications of functional entities. In this method, we use intermediate languages STR(L) and STR/D(L). They have the same syntax as STR and STR/D, but they specify the local specifications of one functional entity.

Specification generation of functional entities consists of the following three steps:

Step 1 Event assignments to functional entities:

The events described in STR rules are manually assigned to functional entities where they truly occur.

Step 2 STR(L) and STR/D(L) rule generation from the original STR and STR/D rules, and event assignments:

First we compare conditions described by the initial-state, event and next-state in an STR rule and position-designations in STR/D rules. If the conditions of the STR rule match those of the STR/D rules, we combine these rules to obtain new global state transition rules. The obtained global state transition rules consist of four elements: the revised initial-state, event, task-designation, and the revised next-state.

Next we divide each obtained global state transition rule into local state transition rules for each functional entity used in executing the tasks in the task-designation; however,

the states of the generated local state transition rules specify global states originated in the STR and STR/D rules. In this division, communication actions are divided into two types: send action and receive action. The event assigned to one of the functional entities in Step 1 is specified in the assigned functional entity.

Finally, we can generate STR(L) and STR/D(L) rules from the obtained local state transition rules. In this generation, the send action generates a new STR/D(L) rule for the appropriate entity, and the receive action generates the event of a new STR(L) rule for the appropriate entity.

Step 3 Functional entity specification generation:

From the generated STR(L) and STR/D(L) rules for each entity, an FSM based entity specification is generated by using the same method in the synthesis of process specification in the protocol synthesis method in Chapter 4.

The above functional entity specification generation can be applied when primitive send and receive actions can be extracted from the tasks to be executed in functional entities.

8.2 Application

8.2.1 Universal Personal Telecommunication

UPT (Universal Personal Telecommunication) [44] permits access to telecommunication services with personal mobility. Each UPT user has a unique UPT number. When a UPT user initiates or receives a call, the access is verified by a check of the UPT number and authentication code. If the authentication is verified, the user can proceed to procedure identification.

UPT services are implemented on the functional model shown in Fig. 1.7. In Fig. 1.7 the Functional Entities (FEs) have the following meanings:

FE1 Originating CCAF
FE2 Originating CCF; associated with SSF
FE3 Transit CCF
FE4 Terminating CCF
FE5 Terminating CCAF
FE6 SCF
FE7 SDF(l) (SDF in the local network)
FE8 SRF
FE9 SDF(h) (SDF in the home network)
where the terms are as follows:
SSF Service Switching Function
SRF Specialized Resource Function
CCF Call Control Function
CCAF Call Control Agent Function
SCF Service Control Function
SDF Service Data Function

This functional model cannot be modeled by the layered architecture described in Fig. 2.1; however, each functional entity can be modeled by the layered architecture. The functional model cannot be observed from outside a communications system. Therefore, we synthesize

the process specification incrementally. First, the "service specifications" of each functional entity are obtained from STR rules, which specify terminal behaviors, and from STR/D rules. Process specifications are then generated from the obtained service specifications of functional entities.

The following is an outline of the steps required for a UPT user to access a UPT service and undergo identification and authentication:

1. Access code input by UPT user
2. Recognition of access code, suspension of call processing in CCF, connection of SRF (Establish Temporary Connection)
3. Prompt and response for user identification (input UPT number)
4. Prompt and response for user authentication (input authentication code)
5. UPT user's service provider provides authentication check and sends result
6. Decision:
   - if successful, proceed to procedure identification
   - if unsuccessful and more attempts allowed, advise user of failure and restart at 3
   - if unsuccessful and no more attempts allowed, advise and release call.

Figure 8.1 shows the information flow for the procedure of "access, identification and authentication". There are two other information flows involved in the above actions: "authentication rejection and retry" and "maximum retries reached".

8.2.2 STR Description of UPT

Figure 8.2 shows the STR description for the information flows of access, identification and authentication; retry; and, maximum retries reached. This description introduces new variables to denote UPT users. The variables declared by "Terminal" denote terminals as before; variables declared by "User" are used for UPT users. In Fig. 8.2, the UPT user gets access through terminal "A", "U" denotes the user's UPT number, and "V" denotes the other users' UPT numbers.

STR rules **r1**, **r2**, **r3** express the information flow in Fig. 8.1. Rule **r4** expresses the sequence of authentication retries performed because of a wrong authentication code. Rule **r5** expresses the sequence when the retry limit is exceeded.

The following gives the meaning of each of the state primitives and events in Fig. 8.2:

State primitives

dial-tone(A) represents a state where a UPT service initiation request can be received.

ident(A) represents a state where a UPT number can be received.

auth(A,U) represents a state where the authentication code for UPT number "U" at terminal "A" can be received.

success(A) represents the authentication succeeded state.

fail(A) represents the authentication failed state resulting from the authentication retry limit being exceeded.

m_limit(A) holds when the authentication retry through terminal "A" exceeds the limit.

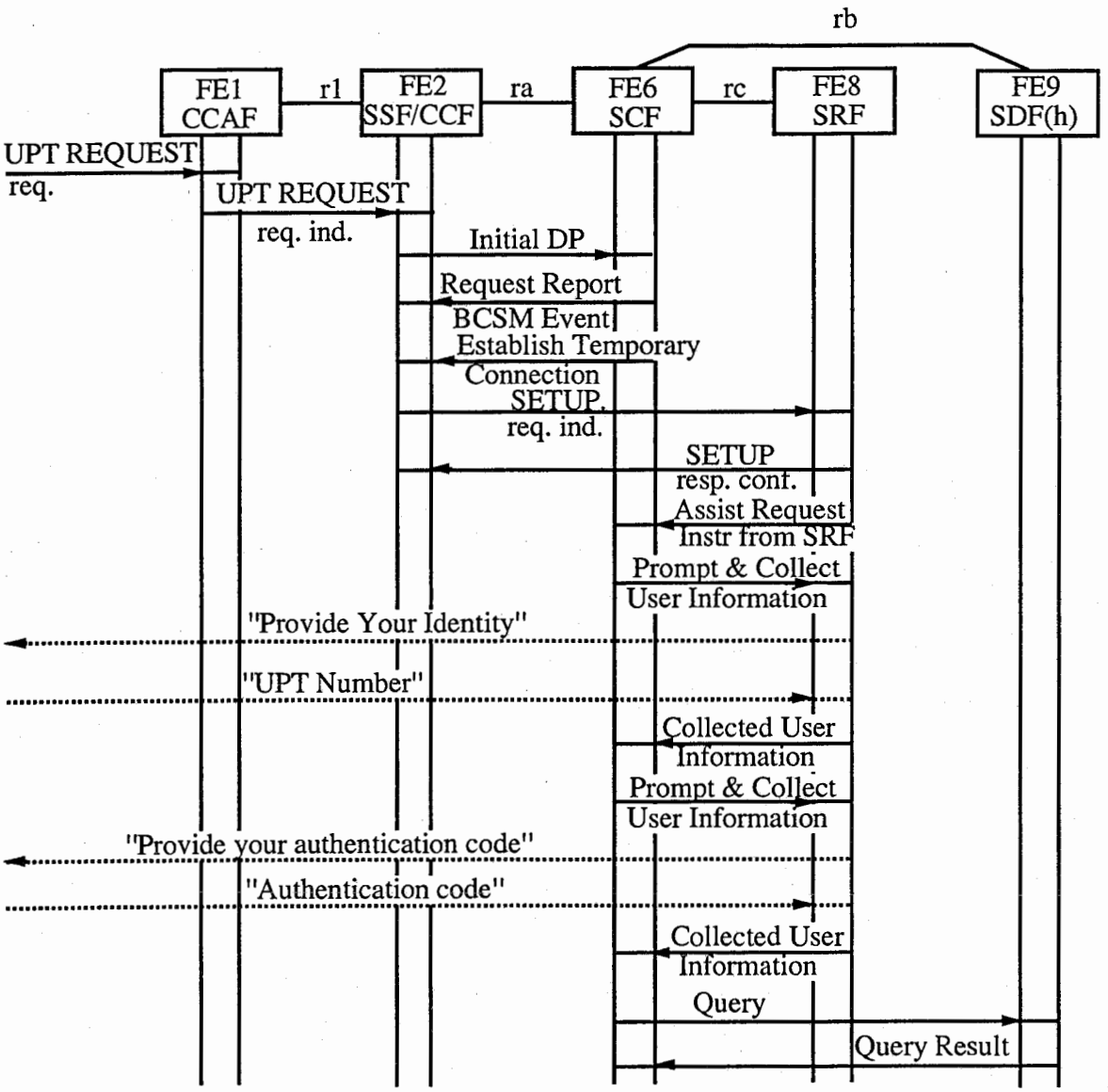


Figure 8.1: Access, identification and authentication.

```
Terminal A;
User U, V;
r1) dial-tone(A)
        uptreq(A):          ident(A).
r2) ident(A)
        idnumber(A,U):      auth(A,U).
r3) auth(A,U)
        acode(A,U):         success(A,U).
r4) auth(A,U)
        acode(A,V):         ident(A).
r5) auth(A,U), m_limit(A)
        acode(A,V):         fail(A).
```

Figure 8.2: STR description of authentication in UPT.

Events

uptreq(A) represents the UPT service initiation request.

idnumber(A,U) represents an event for which a UPT identification number is received. "U" denotes the received UPT user's identification number.

acode(A,U) represents reception of the authentication code for user "U" at terminal "A".

8.2.3 STR/D Description

Figure 8.3 shows the supplementary specification needed to implement the STR rules in Fig. 8.1 on the functional model in Fig. 1.7. In addition to the declaration of the STR description, the functional entities defined in Fig. 1.7 are declared by "Entity". Tasks described in the "task-designation" are provided as routines.

If there are two or more entities described as parameters of a task, then the task includes communications between the described entities. Tasks are carried out in the described order when more than one task is described in a single "task-designation". Therefore, the original order of communications is preserved.

8.2.4 Stepwise Refinement of UPT

We apply the stepwise refinement method to generate functional entity specifications from the specifications in Fig. 8.1 and Fig. 8.3.

Step 1 The events described in the rules in Fig. 8.1 are manually assigned to functional entities as follows:

uptreq(A)	:	CCAF
idnumber(A,U)	:	SRF
acode(A,C)	:	SRF

Step 2 We now generate new STR(L) and STR/D(L) rules according to the assignment obtained in Step 1. "s1(A)", "s2(A)" are newly generated state primitives. The following rules are the STR and STR/D rules for the entity SCF. Note that each generated STR rule specifies a local state transition of one functional entity.

```
Terminal A;
User U;
Entity CCAF, SSF/CCF, SCF, SRF;

transition(-(dial-tone(A)) +(ident(A)))
{ UptReq(CCAF, SSF/CCF);
  InitialDP(SSF/CCF, SCF);
  ReqReport(SCF, SSF/CCF);
  TempConnect(SCF, SSF/SCF);
  SetupReqInd(SSF/CCF, SRF);
  SetupRespConf(SRF, SCF);
  AssistReq(SRF, SCF);
  PromptCollect(SCF, SRF, "Provide your
      identity"); }
input(event number(A, U))
{ CollectedUserInf(SRF, SCF);
  PromptCollect(SCF, SRF, "Provide your
      authentication code"); }
transition(-(auth(A, U)) +(success(A)))
{ CollectedUserInf(SRF, SCF); }
transition(-(auth(A, U)) +(ident(A)))
{ PromptCollect("Wrong authentication,
      please retry.
      Provide your authentication code"); }
transition(-(auth(A, U)) +(fail(A)))
{ PromptCollect("Retry limit exceeded.
      Your line is now blocked.
      Please hang up."); }
```

Figure 8.3: STR/D rules for authentication.

```
dial-tone(A) InitialDP(A): s1(A).
s1(A) AssistReq(A): s2(A).
s2(A) CollectedUserInf(A): ident(A).

transition(-(dial-tone(A)) +(s1(A)))
{ send(SSF/CCF,ReqReport);
  send(SSF/CCF,TempConnect); }
transition(-(s1(A)) +(s2(A)))
{ send(SRF,PromptCollect,
      "Provide your identity"); }
transition(-(s2(A)) +(ident(A)))
{ send(SRF,PromptCollect,
      "Provide your authentication code"); }
```

Step 3 Finally, we generate entity specifications from these generated STR(L) and STR/D(L) rules.

Chapter 9

Completing Protocols

The formal design of protocols produces reliable protocols through the automatic detection of protocol errors. In this chapter we present an error-free protocol synthesis method by completing protocols synthesized from service specifications described by message sequence charts.

9.1 Protocol Model

9.1.1 Protocol and Service Specifications

Protocol specifications are descriptions utilizing concurrent processes made up of sets of processes that can communicate. Sets of processes that define protocol specifications are called objects. A process is represented by a limited state transition machine with one initial state (and also with a final state) [47]. Each process has receive channels that can receive messages from other processes in FIFO (First-In-First-Out) order. A channel from process p to process q is expressed as ch_{pq}. The unit of action executed by a process is called an event, and a process produces state transitions by executing events.

There are four types of events: send events, receive events, output events, and input events. Send events are represented by the form $-q(m)$. When process p executes $-q(m)$, message m is attached to the end of channel ch_{pq}. Receive events are represented by the form $+p(m)$. When process q executes $+p(m)$, message m at the beginning of channel ch_{pq} is removed. An output event $-(m)$ signifies that the process executed by this event is outputting message m to the outside, and input event $+(m)$ means that the process executed by this event is inputting message m from the outside.

For nondeterministic branches to be excluded from process specifications and for programs to be automatically generated from process specifications, transitions from send events and output events are not included in branches deriving from multiple transitions obtained from the same state [9].

Service specifications representing requirements with respect to objects are expressed using message sequences. A message sequence is the set of event sequences for each process.

Definition 4 (Service specification) *Service specifications with respect to objects consist of sets of message sequences that are written in the following syntax:*

`object` *obj* $\{ p_i = seq_i \mid i = 1, \cdots, N \}$,

obj: object name

p_i*: process name*

$$\texttt{object}\ ms\ \{p = +(req) - q(1) + q(2) - (ack),$$
$$q = +p(1) - (called) + (ok) - p(2)\}$$

Figure 9.1: Example of a message sequence.

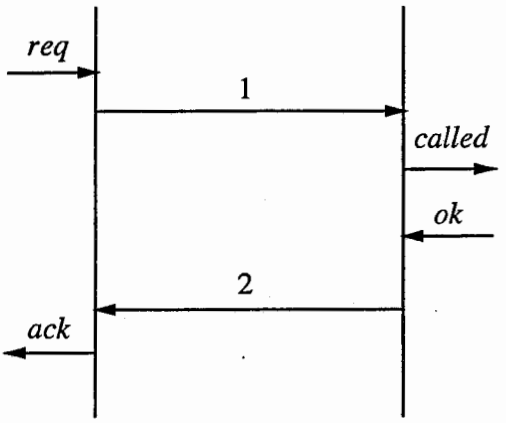


Figure 9.2: Message sequence chart of Fig. 9.1.

seq_i: event sequence

In addition to the four types of events already defined, there are also quasi-events that represent subobject calls. Subobjects are defined as message sequences in which the object name has been replaced by a subobject name.

In all event sequences, events to the left are executed before events to the right. With respect to subobject call events, events in the same process that called a subobject are executed from left to right; then after the events in the subobject have finished being executed, the event immediately following the subobject call event is executed next. At the time a subobject is called and the subobject has finished being executed, the channel over which the process was received must be completely empty.

A diagram representing a message sequence is called a message sequence chart. Figure 9.1 is an example of a message sequence representing the service specification for an object *ms* made up of two processes p and q. The corresponding message sequence chart is shown in Fig. 9.2.

The conditions under which service specifications are converted to a protocol are as follows.
C1: The order in which events are executed in the service specifications is preserved.
C2: No event not already in the service specifications is added.

Protocols can be synthesized out of sets of message sequences that satisfy these conditions [9]. Figure 9.4 shows a protocol synthesized from service specifications based on the message sequences shown in Figs. 9.1 and 9.3. State 0 is the initial and also the final state of each process.

9.1.2 Behaviors in Protocols

After defining the executability of events, we next define behaviors that can occur in protocols. Let process p be in state s. When event f is allocated to produce a transition from state s to

$$\texttt{object}\ ms\ \{p = +q(3) - (called) + (ok) - q(4),$$
$$q = +(req) - p(3) + p(4) - (ack)\}$$

Figure 9.3: Another message sequence for object ms.

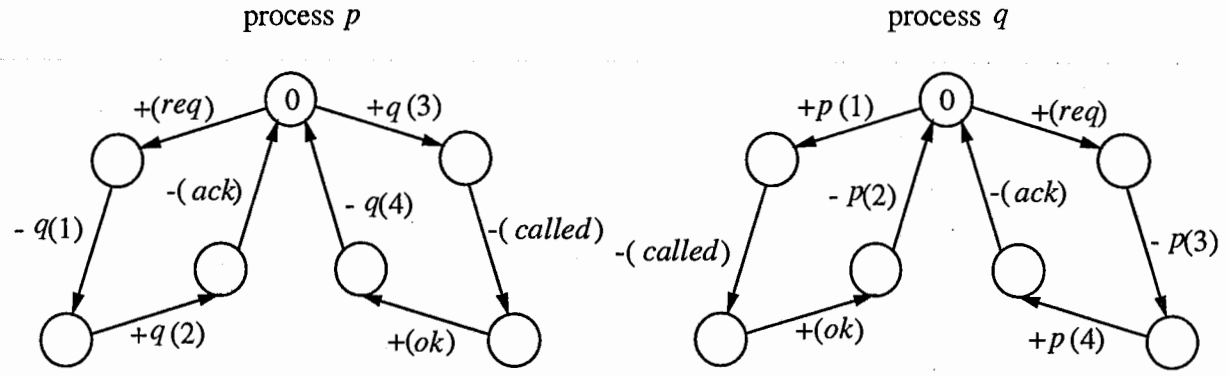


Figure 9.4: Example protocol synthesized from message sequence charts.

state t, the following conditions must be defined in order for event f to be executable. If these conditions are met, process p produces a transition to state t after event f is executed.

- When f is $-(m)$, it is executable without conditions.
- When f is $-q(m)$, it is executable without conditions, and message m is added to the end of ch_{pq}.
- When f is $+(m)$, it is executable if message m is input from the outside.
- When f is $+q(m)$, it is executable if message m is present at the beginning of ch_{qp}; after the event is executed, message m is removed from the beginning of ch_{qp}.

Assume a protocol P. When a given sequence of messages is input to P, the behavior occurring in P will be defined. Regarding all processes included in P, event execution begins from an initial state, and behaviors are defined in terms of the event sequence sets derived for each process.

Definition 5 (Behavior) *A behavior b with respect to a protocol P is defined as a set of process event sequences satisfying the conditions:*

(1) A process event sequence is described as:
$p_i = seq_i$, where p_i is a process name in P and seq_i is an event sequence of process p_i. If $p_i = seq_i$ and $p_j = seq_j$ are two process event sequences in b, then $p_i \neq p_j$.

(2) Let all the processes in P be in their initial states. Then, there is a sequence s of input messages from the outside such that all the events in b can be executable in P according to the above definition of event execution. Once a process returns to its final state, the process can never execute any event in b with respect to s.

$$b = \{p = +(req) - q(1),$$
$$q = +p(1) - (called)\ \}$$

Figure 9.5: Example of a partial behavior.

In the event there are two behaviors b_1 and b_2 and every process event sequence of b_1 is a prefix of the same process event sequence of b_2, then b_1 is called a partial behavior of b_2. In Fig. 9.5, the event sequence set b is a behavior of the object *ms* in Fig. 9.4 and is a partial behavior of the message sequence in Fig. 9.1.

When all processes have reached their final states after executing all event sequences belonging to a behavior, and all channels are empty when the final states are reached, then that behavior is said to be *completed.*

9.2 Definition and Nature of Exceptional Behaviors

9.2.1 Definition of Exceptional Behaviors

In protocols made up of service specifications with multiple message sequences, it is possible for behaviors to occur that do not correspond to the message sequences configuring the service specifications of required specifications; in other words, non-required behaviors.

Let behavior b consist of two elements: event sequence s_1 of process p_1 and event sequence s_2 of process p_2. When a send event included in s_1 and a receive event included in s_2 make up a communication between process p_1 and process p_2, then s_1 and s_2 are defined as being *linked.* This linked relationship is transitive.

Definition 6 (Exceptional behavior) *With respect to service specifications R, when behavior b of a protocol satisfying R meets the following two conditions, then b is said to be an exceptional behavior:*

1 *Behavior b is not a partial behavior of any service specification.*

2 *Regarding event sequences that are elements of behavior b, the following apply:*

   (a) A subset c of b exists such that the elements of c are mutually linked, and c is not a partial behavior of any service specification that is included in R.

   (b) No process in b can execute an event with respect to any arbitrary input from the outside.

Condition 2(a) excludes behaviors from exceptional behaviors that occur when two message sequences with no mutual influence are executed at the same time. The following behavior example illustrates the only exceptional behavior that could occur in the protocol illustrated in Fig. 9.4. The exceptional behavior exhibits the property of 2(b). Process p waits for message 2 from process q, and process q waits for message 4 from process p after executing every event described in ex. This is a deadlock state. Consequently, these two processes cannot execute any events anymore with respect to any arbitrary input from the outside.

$$ex = \{p = +(req) - q(1),$$
$$q = +(req) - p(3)\}$$

9.2.2 Nature of Exceptional Behaviors

Exceptional behaviors occur when a protocol initiates an action in line with a message sequence, and an event is executed that does not belong to the message sequence. The event causing this exceptional behavior is called a causative event.

Definition 7 (Causative event) *Let the set of service specifications be represented by R, and assume that event sequence set c is an exceptional behavior in a protocol synthesized from R. Let d be a partial behavior of c. Then an event f in c is said to be a causative event of exceptional behavior c if the following conditions are satisfied:*

1. *There is a service specification in R that contains d as its partial behavior.*
2. *The behavior derived by executing all events in d and f is not a partial behavior of any service specification in R.*

Let c be an exceptional behavior of protocol P synthesized from service specifications R, and let partial behavior d of c be a partial behavior of $r(\in R)$. Considering that c is the exceptional behavior derived by executing event f after executing all events contained in d, f is one of the following:

- An input event configuring a branch.
- A receive event configuring a branch.
- If e is considered to be an event deriving from an initial state and not configuring a branch with another event, then e is an input event, an output event, or a send event.

Other events besides those discussed above must derive from a state other than an initial state and cannot configure a branch. This clarifies that they cannot be causative events of exceptional behaviors.

9.3 Completing Algorithm

We define how to complete protocols, including exceptional behaviors, and then show the algorithm.

9.3.1 Definition of Completion

Definition 8 (Detection of exceptional behavior) *When there is an event f that may be executed only in exceptional behavior b at process p, if f has been executed at p, it is defined that p detected exceptional behavior b.*

Definition 9 (Protocol completion) *Let b be an exceptional behavior that may happen in protocol P. Exceptional behavior b is called completed if one of the processes in P detects that b is an exceptional behavior and P is modified so that all the processes go back to their final states with empty channels. If every exceptional behavior in P is completed, then protocol P is called completed.*

We note that every process associated with an exceptional behavior can detect the exceptional behavior if one of the processes detects it. This is achieved by exchanging messages between the process that detected the exceptional behavior and other processes.

The matter of what to do after an exceptional behavior is completed depends on the situation, including designer's intention. There are several cases. For example, every process returns to its initial state, returns to its specific state and then proceeds to a succeeding procedure, or proceeds to an error procedure. Therefore we introduce a function to specify a state where a process goes to when an exceptional behavior is completed in the process. In the following method, *state* plays this role.

9.3.2 Completing Method

We clarify a range that is influenced by event addition to complete a protocol.

Definition 10 (Scope influenced by modification) *Let $R = \{r_1, \cdots, r_m\}$ be a set of service specifications and $e = \{p_1 = b_1, \cdots, p_n = b_n\}$ be an exceptional behavior that may happen in the protocol synthesized from R. Let $scope(b_k)$ be the set of service specifications such that a behavior of process p in $r_l (\in R)$ contains b_k as a partial behavior.*

$scope(b_k)$ represents service specifications that are influenced by modifying behavior b_k in p_k. For any pair of $p_k = b_k (\in e)$ and $r_l (\in scope(b_k))$, there is $p_v = b_v (\in s)$ such that $r_l \notin scope(b_v)$. If there are multiple $p_v = b_v$, we determine one of them by selecting the smallest suffix v. We denote $\langle k, l \rangle$ as a suffix determined by suffixes k and l. In the following, however, $\langle k, l \rangle$ is not necessarily the smallest suffix. We have only to determine one suffix.

Since there is an assumption that every receive channel is empty when a subobject is called, we have only to complete every object and subobject. We do not have to take account of the interaction among an object and subobjects. The initial state of a subobject is defined as the state where no event has yet been executed; the final state of a subobject is defined as the state where every process has been executed and is finished, i.e., there is no transition leaving the state.

A reachability analysis can detect exceptional behaviors as possible behaviors in a protocol that does not correspond to any service specifications [65]. Completing a protocol requires completing every latent exceptional behavior in the protocol. We note that both service specifications and exceptional behaviors may be modified upon completing exceptional behaviors.

Input

Service specifications $R = \{r_1, \cdots, r_l\}$
A synthesized protocol specification $P = \{p_1, \cdots, p_n\}$
Exceptional behaviors $E = \{e_1, \cdots, e_m\}$
$state(e_i, p)$: A state to be reached at process p after completing exceptional behavior e_i.

Output

A completed protocol specification $P' = \{p'_1, \cdots, p'_n\}$.

Function

$new(p)$: Returns a new state that is different from any other state.

Procedure

Assume that exceptional behaviors $e_1, \cdots, e_k$ have been completed. Let $R^{(k)} = \{r_1^{(k)}, \cdots, r_l^{(k)}\}$, $r_j^{(k)} = \{p_1 = a_{j,1}^{(k)}, \cdots, p_n = a_{j,n}^{(k)}\}$ be the service specifications; $E^{(k)} = \{e_1^{(k)}, \cdots, e_m^{(k)}\}$ be the exceptional behaviors; $P^{(k)} = \{p_1^{(k)}, \cdots, p_n^{(k)}\}$ be the protocol specification, at that time. We now complete $e_{k+1}^{(k)}$. Let $e_{k+1}^{(k)} = \{p_1 = c_1^{(k)}, \cdots, p_n = c_n^{(k)}\}$. Events of a message sequence $e_{k+1}^{(k)}$ in the protocol $P^{(k)}$ are executed according to the definition without input from the outside. Let st_{p_i} be the state of process p_i $(i = 1, \cdots, n)$, and $cont_{ij}$ be contents of the channel from process p_i to p_j $(i = 1, \cdots, n,\ j = 1, \cdots, n,\ i \neq j)$ when the execution has finished. We use $new(p)$ if a new state is needed in process p on the way to the completion of $e_{k+1}^{(k)}$. When $e_{k+1}^{(k)}$ is completed, we change the reached state to $state(e_{k+1}^{(k)}, p)$.

Case 1 Neither st_{p_i} is the final state of process p_i $(i = 1, \cdots, n)$, and there is at least one nonempty channel for each process.

Append receive events to each process p_i to receive every remaining message in each channel.

Case 2 Every st_{p_i} is the final state of process p_i $(i = 1, \cdots, n)$ and every $cont_{ij} (1 \leq i \leq n, 1 \leq j \leq n, i \neq j)$ is empty.

Assume that

$$e_{k+1}^{(k)},$$
$$r_j^{(k)} (j = 1, \cdots, l)$$

becomes

$$e_{k+1}^{(k)(i)} = \{p_1 = c_1^{(k)(i)}, \cdots, p_n = c_n^{(k)(i)}\}$$
$$r_j^{(k)(i)} = \{p_1 = a_{j,1}^{(k)(i)}, \cdots, p_n = a_{j,n}^{(k)(i)}\}$$

when $p_1 = c_1^{(k)}$, $\cdots$, $p_i = c_i^{(k)}$ have been completed. We modify service specifications in $scope(c_{i+1}^{(k)(i)})$ and the exceptional behavior $e_{k+1}^{(k)(i)}$ so that they give different event sequences in process p_{i+1}. Let $g_{i,j}$, $h_{i,j}$ $(i = 1, \cdots, n;\ j = 1, \cdots, \#(scope(c_{i+1}^{(k)(i)})))$ be new messages, where $\#(scope(c_{i+1}^{(k)(i)}))$ is the number of elements of $(scope(c_{i+1}^{(k)(i)}))$. For each $r_u \in scope(c_{i+1}^{(k)(i)})$, we choose a service specification $r_v \in scope(c_{\langle i+1,u \rangle}^{(k)(i)})$. Note that two event sequences of process $p_{\langle i+1,u \rangle}$ in $e_{k+1}^{(k)(i)}$ and r_u are different. Then we modify them as follows.

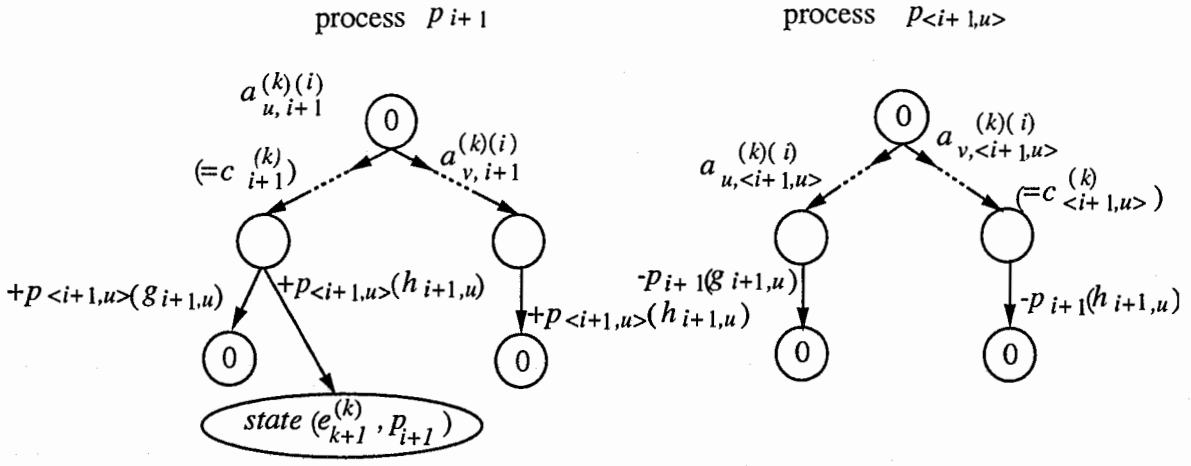


Figure 9.6: Separation of en exceptional behavior from a service specification.

$$
\begin{aligned}
r_u = \{ \quad p_1 &= a^{(k)(i)}_{u,1}, \cdots, \\
p_{i+1} &= a^{(k)(i)}_{u,i+1} + p_{\langle i+1,u\rangle}(g_{i+1,u}), \cdots, \\
p_{\langle i+1,u\rangle} &= a^{(k)(i)}_{u,\langle i+1,u\rangle} - p_{i+1}(g_{i+1,u}), \cdots, \\
p_n &= a^{(k)(i)}_{u,n}\} \\
r_v = \{ \quad p_1 &= a^{(k)(i)}_{v,1}, \cdots, \\
p_{i+1} &= a^{(k)(i)}_{v,i+1} + p_{\langle i+1,u\rangle}(h_{i+1,u}), \cdots, \\
p_{\langle i+1,u\rangle} &= a^{(k)(i)}_{v,\langle i+1,u\rangle} - p_{i+1}(h_{i+1,u}) \cdots, \\
p_n &= a^{(k)(i)}_{v,n}\} \\
e^{(k)(i)}_{k+1} = \{ \quad p_1 &= c^{(k)}_1, \cdots, \\
p_{i+1} &= c^{(k)}_{i+1} + p_{(i+1,u)}(h_{i+1,u}), \cdots, \\
p_{\langle i+1,u\rangle} &= c^{(k)}_{\langle i+1,u\rangle} - p_{i+1}(h_{i+1,u}), \cdots, \\
p_n &= c^{(k)}_n\}
\end{aligned}
$$

We reflect the modification to the protocol as shown in Fig. 9.6. Then process p_{i+1} detects the exceptional behavior at state $state(e^{(k)}_{k+1}, p^{(i+1)})$.

Case 3 Other exceptional behaviors.

Every state of process p_i(i=1,$\cdots$,n) is classified into one of the following four cases, and there is at least one process classified into (b) or (c).

(a) st_{p_i} is the final state, and $cont_{ij} = \emptyset$ ($i \neq j, j = 1,\cdots,n$).

(b) st_{p_i} is the final state, and there is at least one channel $ch_{p_ip_j}$ such that $cont_{ij} \neq \emptyset$.

(c) st_{p_i} is not the final state, and $cont_{ij} = \emptyset$ ($i \neq j, j = 1,\cdots,n$).

(d) st_{p_i} is not the final state, and there is a channel $ch_{p_ip_j}$ such that $cont_{ij} \neq \emptyset$.

If a process classified into (b) or (c) detects an exceptional behavior, the processes classified into (a) can be completed as in Case 2. In other words, the processes classified into (a) receive a message from a process classified into (b) or (c), and detect the exceptional behavior. The

procedure in Case 1 can be applied to the processes classified into (d). Consequently, we have only to modify processes classified into (b) or (c) to detect exceptional behaviors and go to the specified states with empty $cont_{ij}$.

Let p_{i+1} be a process classified into (b) or (c). Assume that

$$e_{k+1}^{(k)},$$
$$r_j^{(k)}(j = 1, \cdots, l)$$

becomes

$$e_{k+1}^{(k)(i)}=\{p_1 = c_1^{(k)(i)}, \cdots, p_n = c_n^{(k)(i)}\}$$
$$r_j^{(k)(i)}=\{p_1 = a_{j,1}^{(k)(i)}, \cdots, p_n = a_{j,n}^{(k)(i)}\}$$

when $p_1 = c_1^{(k)}, \cdots, p_i = c_i^{(k)}$ has been completed. Let g_{ij} $(i = 1, \cdots, n; j = 1, \cdots, \#(scope(c_{i+1}^{(k)(i)})))$ be new messages. Then we modify service specifications and the exceptional behavior as follows.

- When p_i is classified into (b):

  Let $d_{i+1}(= +p_{j_1}(h_1) \cdots + p_{j_f}(h_f))$ be an event sequence to receive all messages remaining in the channels of p_i. We modify $e_{k+1}^{(k)(i)}$ as follows.

$$\begin{aligned} e_{k+1}^{(k)(i)} = \{ \quad p_1 &= c_1^{(k)(i)}, \cdots, \\ p_{i+1} &= c_{i+1}^{(k)(i)}\ d_{i+1}, \cdots, \\ p_n &= c_n^{(k)(i)}\} \end{aligned}$$

  For each $r_j \in scope(c_{i+1}^{(k)(i)})$, we let $pred(p_{\langle i+1,j\rangle})$ be the event sequence of process $p_{\langle i+1,j\rangle}$ when r_j is executed and the event sequence of process p_{i+1} has become $c_{i+1}^{(k)(i)}$, and $succ(p_{\langle i+1,j\rangle})$ be the remaining event sequence of process $p_{\langle i+1,j\rangle}$. Then we modify $r_j^{(k)(i)}$ as follows.

$$\begin{aligned} r_j^{(k)(i)} = \{ \quad p_1 &= a_{j,1}^{(k)(i)}, \cdots, \\ p_{i+1} &= c_{i+1}^{(k)(i)} + p_{\langle i+1,j\rangle}(g_{i+1,j}), \cdots \\ p_{\langle i+1,j\rangle} &= pred(p_{\langle i+1,j\rangle}) - p_{i+1}(g_{i+1,j})\ succ(p_{\langle i+1,j\rangle}), \cdots, \\ p_n &= a_{j,n}^{(k)(i)}\} \end{aligned}$$

  We reflect the modification to the protocol as shown in Fig. 9.7.

- When p_i is classified into (c):

  For each $r_j \in scope(c_{i+1}^{(k)(i)})$, we modify as follows.

$$\begin{aligned} e_{k+1}^{(k)(i)} = \{ \quad p_1 &= c_1^{(k)(i)}, \cdots, \\ p_{i+1} &= c_{i+1}^{(k)(i)} + p_{\langle i+1,j\rangle}(g_{i+1,j}), \cdots, \\ p_{\langle i+1,j\rangle} &= pred(p_{\langle i+1,j\rangle}) - p_{i+1}(g_{i+1,j})\ succ(p_{\langle i+1,j\rangle}), \cdots, \\ p_n &= c_n^{(k)(i)}\} \\ r_j^{(k)(i)} = \{ \quad p_1 &= a_{j,1}^{(k)(i)}, \cdots, \\ p_{i+1} &= pred(p_{i+1}) + p_{\langle i+1,j\rangle}(g_{i+1,j})\ succ(pred(p_{i+1})), \cdots, \\ p_{\langle i+1,j\rangle} &= pred(p_{\langle i+1,j\rangle}) - p_{i+1}(g_{i+1,j})\ succ(p_{\langle i+1,j\rangle}), \cdots, \\ p_n &= a_{j,n}^{(k)(i)}\} \end{aligned}$$

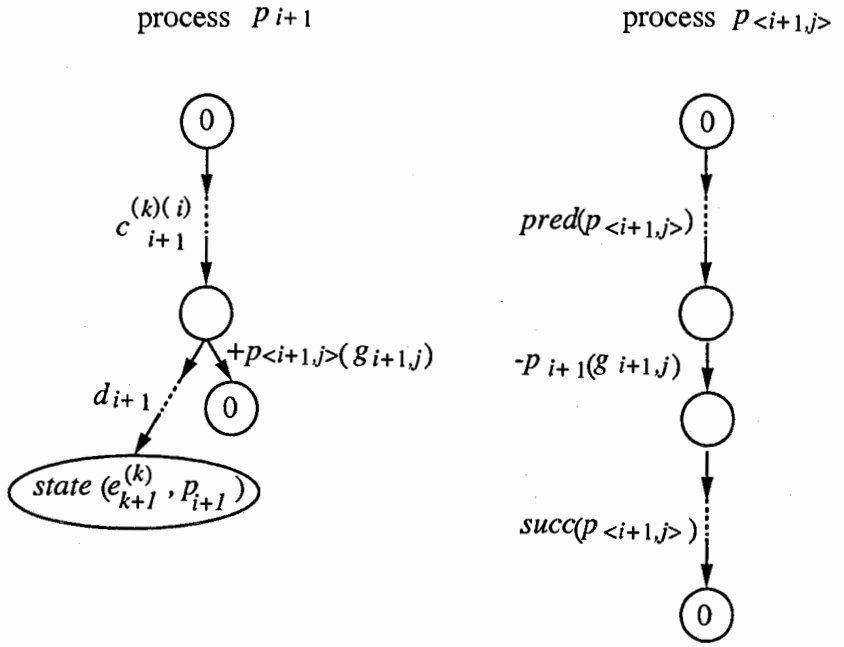


Figure 9.7: Separation of an exceptional behavior classified into (b) from a service specification.

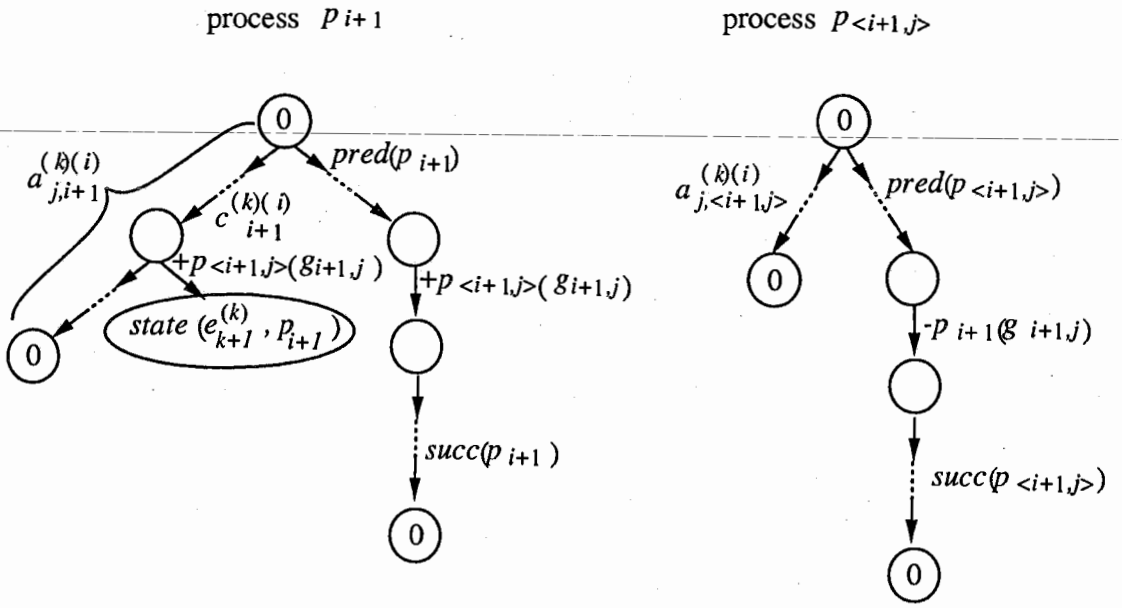


Figure 9.8: Separation of an exceptional behavior classified into (c) from a service specification.

We reflect the modification to the protocol as shown in Fig. 9.8.

The above procedure completes a synthesized protocol P from the service specifications R when a set of exceptional behaviors E is given. Exceptional behaviors classified into Case 1 can be completed without modifying any service specifications. Completing other exceptional behaviors need modification of service specifications. Exceptional behaviors classified into Case 3(a) are not handled in conventional protocol completion methods without using service specifications explicitly.

9.4 Application

We apply the protocol completion algorithm to X.227 [66]. First we briefly explain X.227. X.227 is one of a set of Recommendations produced to facilitate the interconnection of information processing systems and specifies the protocol for the association-service-element for application-association control: the Association Control Service Element (ACSE). The ACSE provides services for establishing and releasing application-associations. The protocol is governed by the use of the presentation-service (X.216) and the session-service (X.215), however, we take account of only X.227 and omit some abnormal procedures. Consequently, the service specifications and the resultant synthesized protocol in this example do not exactly coincide with Recommendation X.227.

9.4.1 Premise

The protocol completion algorithm is supposed to be applied under the situation:

1. A designer describes message sequences as service specifications.
2. The designer then synthesizes a protocol from the specifications. This synthesis can be automated [9]. If the protocol includes an exceptional behavior, then the designer completes the protocol by applying the algorithm.

Therefore, the above-mentioned situation cannot be applied in the case of an already completed protocol. Then, assuming the following protocol design steps, we show that the objective protocol specification is obtained by simulating protocol completion.

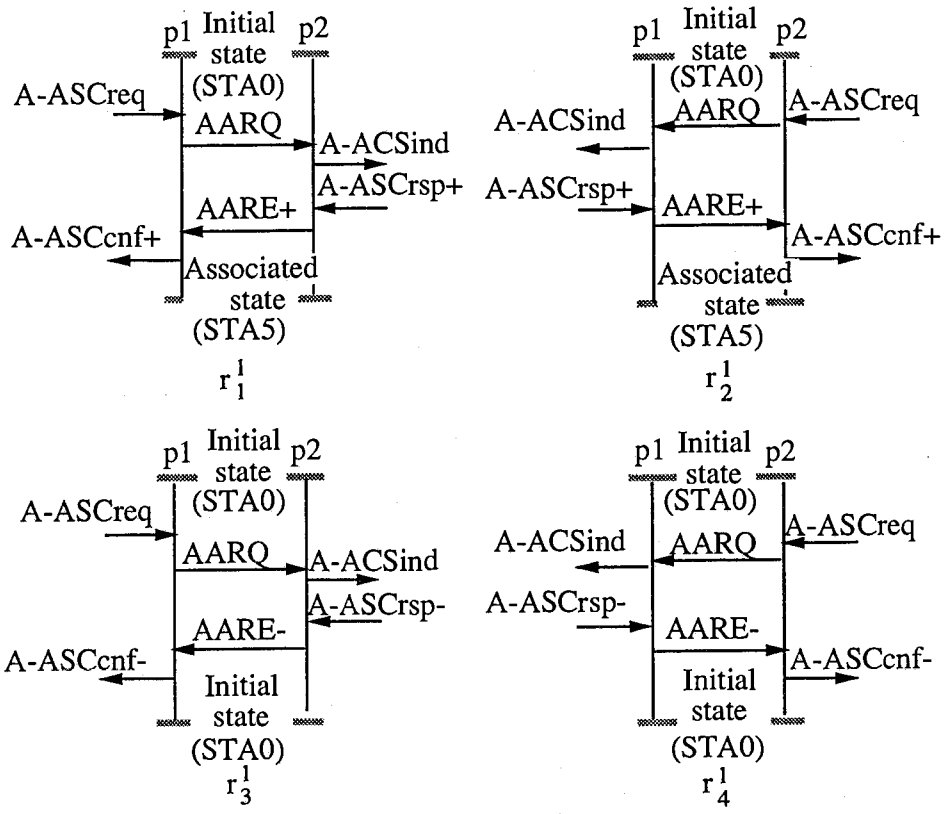
1. We extract primary sequences of X.227 given as a standard and assume that they constitute the service specifications.
2. We obtain the protocol specification of X.227 by completing the protocol synthesized from the service specifications.

9.4.2 Completing Process

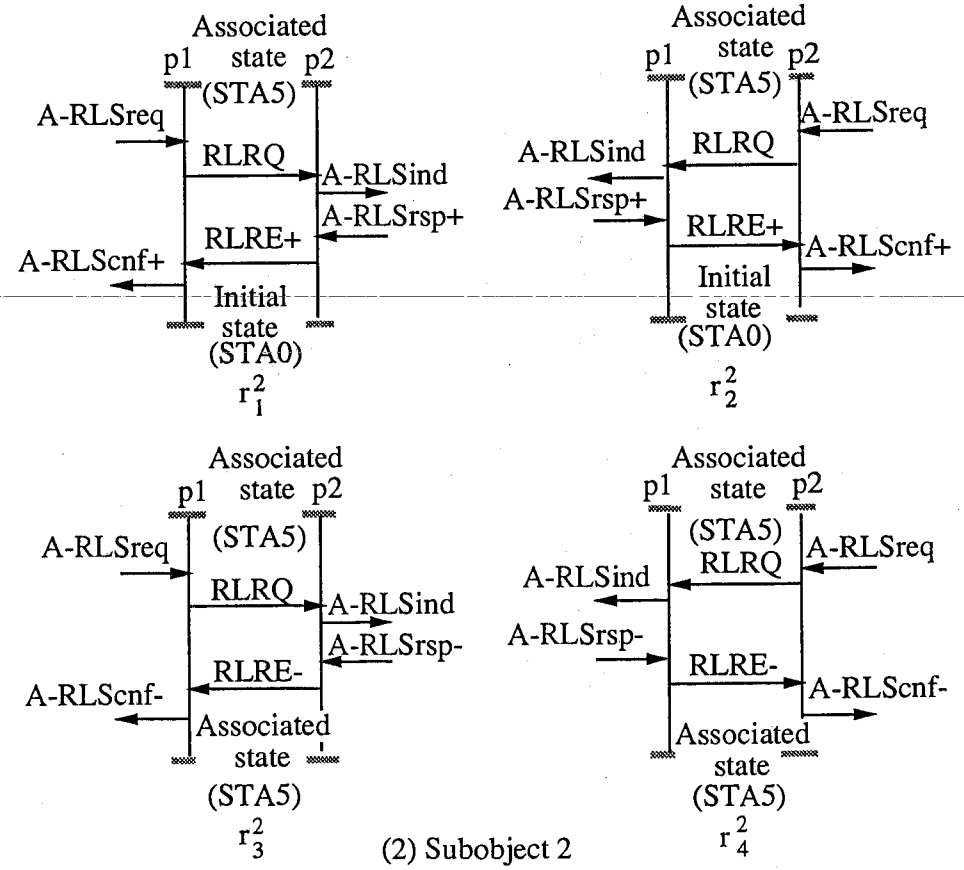
Preliminary

We assume the next two subobjects as service specifications. Figure 9.9 illustrates the subobjects.

1. Subobject 1: A sequence from the initial state to the associated state. Let the service specifications be $R^1 = \{r_1^1, r_2^1, r_3^1, r_4^1\}$.



(1) Subobject 1



(2) Subobject 2

Figure 9.9: Service specifications of X.227.

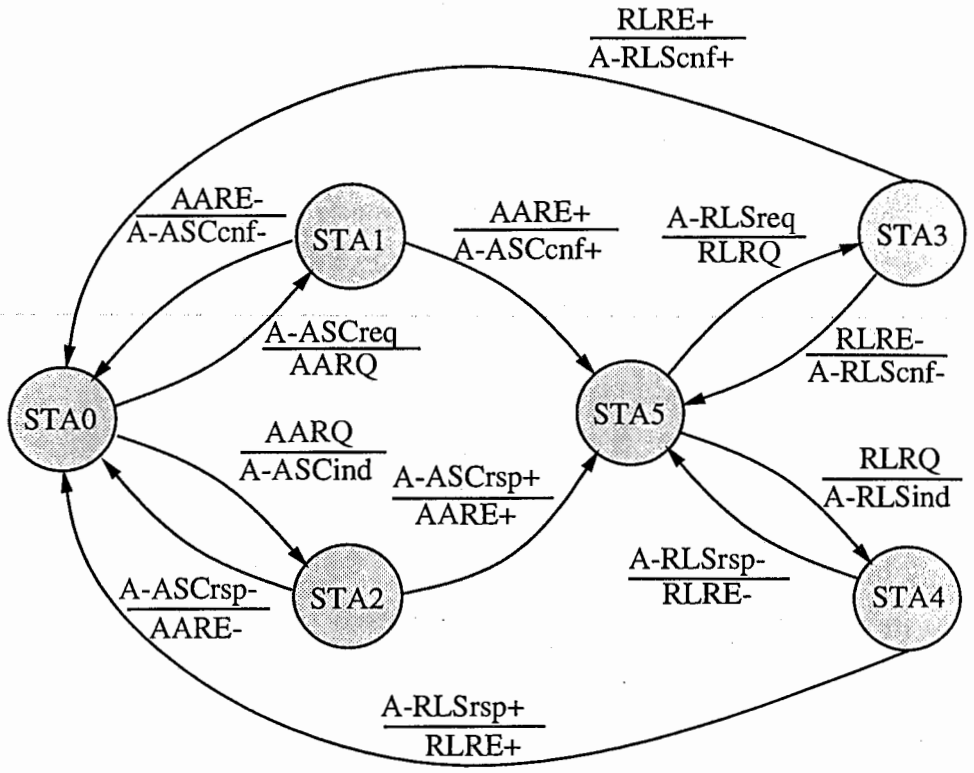


Figure 9.10: Synthesized protocol from service specifications in Fig. 9.9.

2. Subobject 2: A sequence from the initial state to the associated state. Let the service specifications be $R^2 = \{r_1^2, r_2^2, r_3^2, r_4^2\}$.

We can synthesize the protocol in Fig. 9.10 from the service specifications. In this figure we use an abbreviated notation to represent events. A message over a bar abbreviates a receive event or an input event. If the message has the prefix "A-", then the message is an abbreviation of the input event of it, and other messages over bars are abbreviations of receive events of them. A message under a bar abbreviates a send event or an output event. If the message has the prefix "A-", then the message is an abbreviation of the output event of it, and other messages under bars are abbreviations of send events.

Completing the protocol

(1) Subobject 1

STEP1: Detection of exceptional behaviors

The protocol includes an exceptional behavior that occurs when A-ASCreq is inputted to processes $p1$ and $p2$ at the same time. Then AARQ is received neither by $p1$ nor by $p2$ as shown in Fig. 9.11. Let these states be st_{p1}^1 and st_{p2}^1.

STEP2: Classification of exceptional behaviors

Since neither st^1_{p1} nor st^1_{p2} is the final state, and there are non-empty channels at both $p1$ and $p2$, the exceptional behavior is classified into Case 3(a) in the previous section.

STEP3: Completion (Fig. 9.11)

- Addition of a receive event for coping with unspecified reception
  Add receive events of AARQ to states st^1_{p1} and st^1_{p2}.
- Decision of additional sequences and states to be reached
  This process is a designer-dependent matter, and there are various procedures. In this example, each process informs its upper layer and processes exchange ABRT with others. Finally each process goes back to its initial state.

(2) Subobject 2

STEP1: Detection of exceptional behaviors

The protocol includes an exceptional behavior that occurs when A-RLSreq is inputted to processes $p1$ and $p2$ at the same time. Then RLRQ is received neither by $p1$ nor by $p2$ as shown in Fig. 9.11. Let these states be st^2_{p1} and st^2_{p2}.

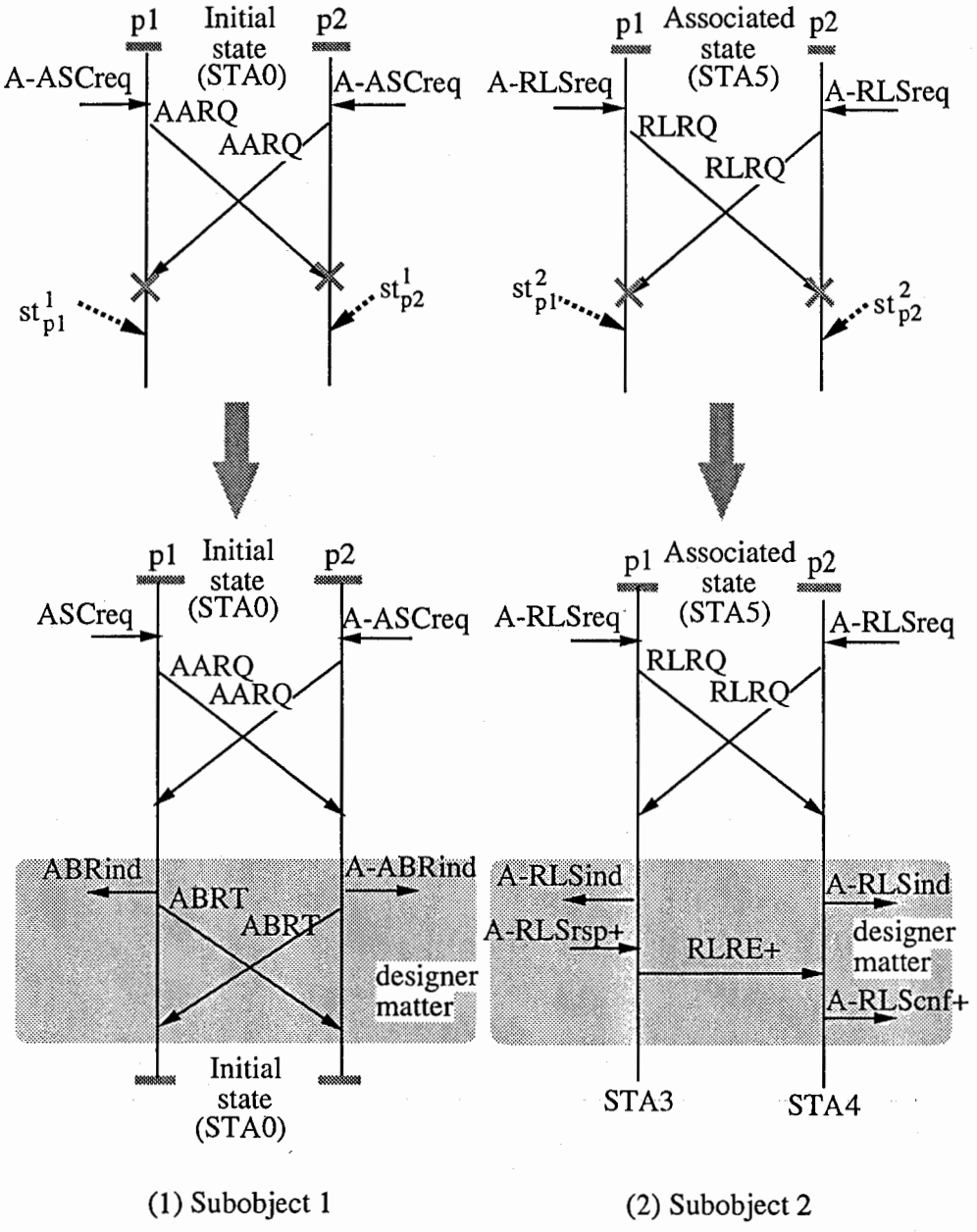
STEP2: Classification of exceptional behaviors

Since neither st^2_{p1} nor st^2_{p2} is the final state, and there are non-empty channels at both $p1$ and $p2$, the exceptional behavior is classified into Case 3(a) in the previous section.

STEP3: Completion (Fig. 9.11)

- Addition of a receive event for coping with unspecified reception.
  Add receive events of RLRQ to states st^2_{p1} and st^2_{p2}.
- Decision of additional sequences and states to be reached.
  This process is also a designer-dependent matter, like in the case of subobject 1. Here, each process informs its upper layer and process $p1$ sends $p2$ RLRE+. Finally process $p1$ goes back to STA3 and process $p2$ goes back to STA4.

The above steps produce complete service specifications that contain additional completed sequences and the original service specifications. It follows that we can get a completed protocol as illustrated in Fig. 9.12 that contains three new states STA6, STA7 and STA8. We note that STA8 is not defined in X.227. This state is created because ABRT can be received at any state except STA0 in the original X.227, however, we omitted such abnormal procedures in this example.



(1) Subobject 1 (2) Subobject 2

Figure 9.11: Exceptional behaviors in X.227 and their completion.

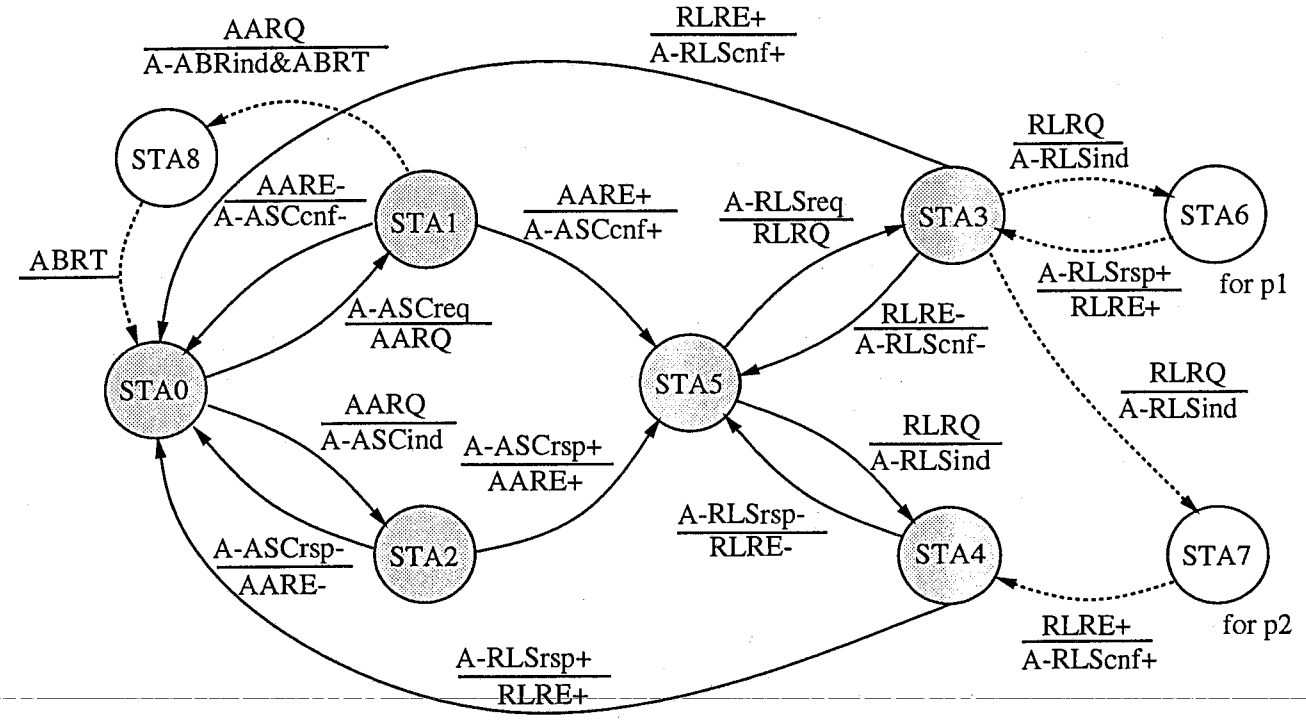


Figure 9.12: X.227 protocol obtained by the completion method.

Chapter 10

Conclusion

We conclude this thesis by summarizing the obtained results. In this thesis we investigated the following three subjects.

1. Transformation from initial incomplete requirements into complete service specifications.
2. Automated software generation from service specifications.
3. Transformation from protocols with errors into complete protocols.

In Chapter 3, we proposed a specification completion method for a rule-based specification language STR. The completed specification agrees with the requirements of the service designer. The method obtains complete service specifications by eliminating errors and supplementing insufficient rules in the initial requirements. Most published works start on the premise that the requirements analysts are different from the users of the software systems. In this method, the users themselves can describe their own requirements rather than requirements analysts. If new rules are necessary, the method generates them by using an abstraction of conventional communication services. The proposed method has a limit, however. If a completely new requirement specification is given, it becomes impossible to generate rules to supplement the incomplete requirement specifications. Future work includes providing a generic domain knowledge.

As far as automated software generation is concerned, we obtained three algorithms. The first two algorithms described in Chapters 4 and 5 synthesize a protocol from service specifications. The algorithms assume a layered architecture which is commonly used for a protocol architecture. They differ in their message exchange methods between protocol entities. The algorithm in Chapter 4 synthesizes a sequential communicating protocol. The algorithm in Chapter 5, in contrast, synthesizes a parallel communicating protocol. A parallel communicating protocol uses more message exchanges than a sequential communicating protocol but usually spends less time for message exchanges. These algorithms are definitely different from protocol synthesis algorithms in published work. In our methods the execution order of events is not specified in service specifications. The execution order of events is usually considered as an implementation dependent factor [64]. Another difference is that the methods in this thesis synthesize protocols implementing distributed algorithms.

In Chapter 6 we defined a detailed specification language STR/D that refines protocol specifications so that they become implementable. The language aims at refining protocol specifications without knowing individual service specifications as best as possible. In an experiment, typical communication services were found to be capable of refinement independent

of the individual rules in Chapter 7. The refinement specifications were dependent on state primitives and events. We implemented several communication services on PBX systems by using a generated software specification. In actual communication networks the protocol between communication systems was specified beforehand. We had a problem determining how to conform the generated software to the protocol.

In order to generate communication software that can be installed on an actual network, we proposed a stepwise refinement method that generates software conforming to the functional model in Chapter 8. This method solves the protocol conformance problem described above. However, the problem of how to increase the ratio of automation still remains.

For the final subject, i.e., to transform erroneous protocols into error-free protocols, we presented a protocol completion method. Exceptional behaviors often happen in communication software which is used to control multiple independent entities. A protocol completing method which resolves undesired states caused by exceptional behaviors has been proposed. There are two cases in which exceptional behaviors are eliminated.

(a) A protocol specification is modified, but service specifications are not.

(b) A protocol specification and service specifications are both modified.

Every exceptional behavior can be completed in the proposed method, though conventional protocol completion methods which do not use service specifications cannot complete some exceptional behaviors. The method has been applied in order to obtain an error-free X.227 protocol from a set of partial specifications of X.227. Future work includes having designers assist in specifying states to be returned to after the resolution of undesirable exceptional behaviors.

These results enable non-experts of communication systems and software to develop communication software semi-automatically, although they have to interact to decide specifications, and the assistance of experts is necessary to transform the obtained protocol specifications into detailed software specifications. The proposed method has the following features:

1. It is possible to obtain complete service specifications that reflect the users' intention from incomplete service specifications.

2. It is possible to generate communication software semi-automatically by only using detailed specifications to define the semantics of primitives and events in STR on a communication system. Furthermore, this detailed knowledge can be described as knowledge.

3. In the case that protocols themselves are specified rather than service specifications, it is possible to obtain complete protocol specifications by describing given protocols by message sequence charts.

Bibliography

[1] Hirakawa, Y. and Takenaka, T., "Telecommunication Service Description Using State Transition Rules", in *Proc. Sixth Int. Workshop on Software Specification and Design* (Como, Italy), pp. 140–147, Oct. 1991.

[2] Cameron, E. J. and Velthuijsen, H., "Feature Interactions in Telecommunications Systems," *IEEE Commun. Magazine*, vol. 31, no. 8, pp. 18–23, Aug. 1993.

[3] Harada, Y., Hirakawa, Y., Takenaka, T. and Terashima, N., "A Conflict Detection Support Method for Telecommunication Service Descriptions", IEICE Trans. Commun., vol. E75-B, no. 10, pp. 986–997, Oct. 1992.

[4] Ohta, T., Takami, K. and Takura, A., "Acquisition of Service Specifications in Two Stages and Detection/Resolution of Feature Interactions," in *The Fourth Telecommunications Information Networking Architecture Workshop* (L'Aquila, Italy), vol. 2, pp. II-173–II-187, Sep. 1993.

[5] Inoue, Y., Takami, K. and Ohta, T., "Automatic Detection of Service Interactions in Telecommunications Service Specifications", in *IEEE International Conference on Communications*, pp. 1835–1841, May 1994.

[6] Zimmermann, H., "OSI Reference Model - the ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Commun*, vol. COM-28, no. 4, pp. 425–432, Apr. 1980.

[7] Menso Appeldom, Roberto Kung, and Roberto Saracco, "TMN + IN = TINA," *IEEE Commun. Magazine*, vol. 31, no. 3, Mar. 1991.

[8] ITU-T, "Message Sequence Chart: Recommendation Z.120," Sep. 1994.

[9] Ichikawa, H., Itoh, M., Kato, J., Takura, A., and Shibasaki, M., "SDE: Incremental Specification and Development of Communications Software," *IEEE Trans. Comput.*, pp. 553–561, Apr. 1991.

[10] Færgemand, O. and Olsen, A., "Introduction to SDL-92," *Computer Networks and ISDN Systems* vol. 29, no. 9, pp. 1143–1167, May 1994.

[11] Bolognesi, T. and Brinksma, E., "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems* vol. 24, pp. 25–59, 1987.

[12] Belina, F. and Hogrefe, D., "The CCITT-Specification and Description Language SDL," *Computer Networks and ISDN Systems*, vol. 16, no. 4, pp. 311–341, 1989.

[13] Faci, M., Logrippo, L. and Stépien, B., "Formal Specification of Telephone Systems in LOTOS: the Constraint-Oriented Style Approach," *Computer Networks and ISDN Systems*, vol. 21, pp. 53–67, 1991.

[14] Drayton, L., Chetwynd, A. and Blair, G., "Introduction to LOTOS through a Worked Example," *computer communications*, vol. 15, no. 2, pp. 71–85, Mar. 1992.

[15] Cameron, E. J., Cohen, D. M., Guithner, T. M., Keese Jr., W. M., Ness, L. A., Norman, C. and Srinidhi, H. N., "The L.0 Language and Environment for Protocol Simulation and Prototyping," *IEEE Trans. Comput.*, vol. 40, no. 4, pp. 562–571, Apr. 1991.

[16] Tsai, J. J. P., Weigert, T. and Jang, H.-C., "A Hybrid Knowledge Representation as a Basis of Requirement Specification and Specification Analysis," *IEEE Trans. on Software Eng.*, vol. 18, No. 12, pp. 1076–1100, Dec. 1992.

[17] Boehm, B. W., "Software Engineering Economics," *IEEE Trans. Software Eng.*, vol. SE-10, no. 1, Jan. 1984.

[18] Reubenstein, H. B. and Waters, R. C., "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Trans. Software Eng.*, vol. SE-17, no. 3, Mar. 1991.

[19] Jarke, M., Bubenko, J., Rolland, C., Sutcliffe, A. and Vassilou, Y., "Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis," *IEEE Int. Symp. on Requirements Engineering* (San Diego, California), pp. 19–31, Jan. 1992.

[20] Rolland, C. and Proix, C., "A Natural Language Approach for Requirements Engineering," *4th Int. CAiSE Conference*, vol. 593, pp. 257–277, 1992.

[21] Puncello, P. P., Torrigiani, P., Pietri, F., Burlon, R., Cardile, B. and Conti, M., "ASPIS: A Knowledge-Based CASE Environment," *IEEE Software*, pp. 58–65, Mar. 1988.

[22] Wrobel, S., "Design Goals for Sloppy Modelling Systems," *Int. J. Man-Machine Studies*, pp. 461–477, vol. 29, no. 4, pp. 461 - 482, Oct. 1988.

[23] Kelly, V. E. and Nonnenmann, U., "Reducing the Complexity of Formal Specification Acquisition, In Automating Software Design," *Automating Software Design*, ed. Lowry, M. R. and McCartney, R. D., AAAI Press, pp. 41–64, 1991.

[24] Probert, R. L. and Saleh, K., "Synthesis of Communication Protocols: Survey and Assessment," *IEEE Trans. Comput.*, vol. 40, no. 4, pp. 468–476, Apr. 1991.

[25] Ichikawa, H. and Takami, K., "Automatic Generation of Communications Software," *The Journal of IEICE*, pp. 522–530, vol. 77, no. 5, May 1994.

[26] Merlin, P. and Bochmann, G. V., "On the Construction of Submodule Specifications and Communication Protocols," *ACM Trans. Programming Languages and Syst.*, vol. 5, no. 1, pp. 1–25, Jan. 1983.

[27] Zafiropulo, P., West, C. H., Rudin, H., Cowan, D. D., and Brand, D., "Towards Analyzing and Synthesizing Protocols," *IEEE Trans. Commun*, vol. Com-28, no. 4, pp. 651–661, Apr. 1980.

[28] Sidhu, D. P., "Protocol Design Rules," in *Protocol Specification, Testing, and Verification*, pp. 283–300, 1982.

[29] Kakuda, Y. and Wakahara, Y., "Component-Based Synthesis of Protocols for Unlimited Number of Processes," in *Proc. IEEE COMPSAC'87*, pp. 721–730, 1987.

[30] Takura, A. and Ichikawa, H., "Completing Protocols According to Requirements," in *Proc. ICCC Symp.* (Beijing, China), pp. 116–119, Sep. 1989.

[31] Takura, A. and Kanai, A., "Completing Protocols Synthesized from Service Specifications," submitted to *IEICE Trans. Commun.*.

[32] Genji, K., Takami, K. and Takenaka, T., "Telecommunication Service Design Support System Using Message Sequence Rules," *IEICE Trans. Commun.*, vol. E75-B, no. 8, pp. 723–732, Aug. 1992.

[33] Bochmann, G. V. and Gotzhein, R., "Deriving protocol specifications from service specifications," in *Communications, Architectures & Protocols, Proc. of the ACM SIGCOMM '86 Symp.*, pp. 148–156, 1986.

[34] Gotzhein, R. and Bochmann, G.V., "Deriving Protocol Specifications from Service Specifications Including Parameters," *ACM Trans. Computer Syst.*, vol. 8, No. 4, pp. 255–283, Nov. 1990.

[35] Chu, P. M. and Liu, M. T., "Synthesizing Protocol Specifications from Service Specifications in FSM Model," *Proc. Comput. Networking Symp.*, pp. 173–182, Apr. 1988.

[36] Saleh, K. and Probert, R., "A Service-Based Method for The Synthesis of Communications Protocols," *Int. J. Mini and Microcomputers*, vol. 12, no. 3, pp. 97–103, 1990.

[37] Kakuda, Y., Nakamura, M. and Kikuno, T., "Automated synthesis of protocol specifications from service specifications with parallelly executable multiple primitives," *IEICE Trans. Fundamentals*, vol. E77-A, no. 10, pp. 1634–1645, Oct. 1994.

[38] Hagihara, K., "Distributed Algorithms," *J. Japanese Society for Artificial Intelligence*, vol 5, no. 4, Jul. 1990.

[39] Garey, M. R. and Johnson, D. S., "Computers and Intractability, A Guide to the Theory of NP-Completeness," NY: W. H. Freeman and Co., 1979.

[40] Takura, A., Kawata, K., Ohta, T. and Terashima, N., "Communication Software Generation Based on Two-Layered Specifications and Execution Environment," in *IEEE GLOBECOM'93* (Houston, USA), pp. 362–368, Dec. 1993.

[41] Dendorfer, C. and Weber, R., "From service specification to protocol entity implementation - An exercise in formal protocol development," in *Protocol Specification, Testing and Verification, XII*pp. 163–177, 1992.

[42] Takura, A., Ohta, T. and Kawata, K., "Task Generation Mechanisms in a Communication Software Generation Systems," in *1993 Asia-Pacific Conf. on Communications*, 2E.2, Aug. 1993.

[43] Sera, T. and Takura, A, "Task Generation for Distributed Functional Model," to appear in *Int'l J. on Artificial Intelligence Tools.*

[44] CCITT, "Draft Recommendation F.851, Universal Personal Telecommunication (UPT) - Service Description," Oct. 1992.

[45] Sera, T, Takura, A. and Ohta, T., "Architecture Refinement for a Distributed Functional Model," in *Proc. of the IASTED Int. Conf.* (Orland, Florida), pp. 173 – 176, Jan. 1996.

[46] Bochmann, G. v., "A general transition model for protocols and communications services," *IEEE Trans. Commun.*, vol. COM-28, no. 4, pp. 643–650, Apr. 1980.

[47] Brand, D. and Zafiropulo, P., "On Communicating Finite-State Machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, Apr. 1983.

[48] Kakuda, Y., "A Recovery Sequence Generation System for Design of Recoverable Protocols," *Trans. IEICE*, vol. E74, no. 6, pp. 1715–1727, Jun. 1991.

[49] Nakamura, M., Takura, A. and Ohta, T., "A Method for Requirements Elicitation Using a Domain Model," in *Task Force on the Engineering of Computer Based Systems*, May 1994.

[50] Ohta, T. and Takura, A., "The Automated Acquisition of Requirements Specifications for Communications Software," in *Proc. of Seventh Int. Conf. Computing and Information (ICCI '95)*, pp. 1009–1021, 1995.

[51] Takura, A, Ueda, Y., Haizuka, T and Ohta, T., "Requirements Acquisition of Communications Services," to appear in *International Communications Conference (ICC '96)*, Jun. 1996.

[52] Takura, A., Sera, T., Ohta, T., "Protocol Synthesis from Service Specifications Described by Graph Rewriting Rules," *RIMS workshop*, Jul. 1995.

[53] Takura, Ohta, T. and Kawata, K., "Process Specification Generation from Communications Service Specifications," *Automated Software Eng.*, no. 2, pp. 167–182, 1995.

[54] Takura, A. and Ohta, T, "Two-Layered Communications Service Specification Description and Program Specification Generation," *Trans. IPS Japan*, vol. 35, no. 5, pp. 1104–1113. May 1995.

[55] Takura, A. and Ohta, T., "Stepwise Telecommunication Software Generation from Service Specifications in State Transition Model," in *1993 Int. Conf. Network Protocols* (Boston, USA), pp. 135–142, Oct. 1994.

[56] Takura, A. and Ohta, T., "Stepwise Refinement of Communications Service Specifications for Conforming to a Functional Model," *IEICE Trans. on Communications*, vol. E77-B, no. 11, pp. 1322–1331, Nov. 1994.

[57] Liu, M. T., "Protocol Engineering," it Advances in Computers, vol. 29, pp. 79–195, 1989.

[58] Takami, K., Harada, Y., Ohta, T. and Terashima, N., "A Visual Design Support System for Telecommunications Service," in *IEEE Phoenix Conf. Computers and Communications*, pp. 593–599, Mar. 1993.

[59] Ueda, Y., Takura, A. and Ohta, T., "A Verification method of Communications System Service Specifications," *Technical Report of IEICE*, KBSE94-44, pp. 25–32, Nov. 1994.

[60] Shibata, K., Hirakawa, Y., Takura, A. and Ohta, T., "Reachability Analysis for Specified Processes in a Behavior Description," *IEICE Trans. Commun.*, pp. 1373–1380, vol. E76-B, no. 11, Nov. 1993.

[61] Sato, M., "Reachability Analysis for Communication Service Specification Descriptions in Global State Rules," *IEICE Trans. Commun.*, pp. 245–251, vol. J78-B-I, Jun. 1995.

[62] Awerbuch, B., "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election," in *Proc. of the 19th ACM Symp. on Theory of Computing*, pp. 230–240, 1987.

[63] Kato, J. and Arakawa, N., "Software Architecture for Automated Communications Software Development," in *Proc. IEE 7th Int. Conf. Software Eng. Telecommun. Switching Syst.*, pp. 29–33, Jul. 1989.

[64] Chandy, K. M. and Misra, J., "Parallel Program Design, A Foundation," Addison-Wesley Publishing Company, 1989.

[65] Itoh, M. and Ichikawa, H., "Protocol Verification Algorithm Using Reduced Reachability Analysis," *Trans. IEICE*, vol. E66, no. 2, pp. 88–93, Feb. 1983.

[66] CCITT Recommendations X.220 - X.229, "Data Communication Networks, Open Systems Interconnection (OSI) Protocol Specifications, Conformance Testing," vol. VIII, Fascicle VIII.5, 1988.