INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa • University of Ottawa

Communications Service Synthesis from Informal Specifications and Sequence Diagrams

By

Nursel Asikhan-Berlinguette

A thesis
submitted to the School of Graduate Studies and Research
in partial fulfilment of the requirements for
the Degree of
Master of Computer Science*

School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada
June 21, 2000

*The Master of Computer Science program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Abstract

Communications Networks consist of layers where each layer provides a service to the layer above. A *service* consists of phases each of which is a sequence of message interactions intended to achieve a specific user goal [Boyc 90]. For example, Transport Service has three phases: connection establishment, data transfer and connection release.

Recent research [Dsou 95] and [Sale 96] has emphasized the need for systematic methods to build services. We propose such a method to build a Global Service FSM (Finite State Machine) from a set of Sequence Diagrams. We use these diagrams to capture a set of constraints on the service: local, end-to-end and concurrency constraints.

The service synthesis method is composed of a series of algorithmic steps. The two major steps are:

i) construction of Phase Scenario Machines (PSM),
ii) coupling of PSMs at the phase boundaries.

The resulting FSM, called a Global Scenario Machine (GSM), is a more precise and complete model of the global service.

This thesis shows that a global service specification, the Global Scenario Machine can be semi-automatically and systematically built from a set of sequence diagrams. More work is needed on tools and verification methods to insure the completeness of this approach and its ease of use, but a realistic case study is used illustrate the feasibility of the approach.

Acknowledgements

I express my deepest appreciation to my thesis supervisor Professor R. L. Probert for his guidance, fruitful discussions support and encouragement throughout my thesis work. Many thanks go to Professor K. Saleh of Kuwait University for his helpful remarks about my thesis research.

I am very grateful to Professors S. Matwin and L. Birta for their support in completing my thesis work, and Ms. N. Lanthier and Ms. M. Moriarty for allowing me enough time to write my thesis. Many thanks also go to Ms. L. Desrochers and Ms. J. Forgues for their great assistance during my thesis work.

I am also very grateful to Professors L. Logrippo and L. Nel, the Examiners, and Professor G. v. Bochmann, the Chairman of the Thesis Examination Committee for their helpful and constructive comments.

I thank my manager Mr. J. Ho very much for allowing me to take time off from my work to complete my thesis.

My deepest respect and love go to my mother, Bahriye Aysel, for encouraging me to further my education abroad, and her loving and courageous attitude and her moral support throughout the years. My very special thanks go to my husband, Paul whose moral support and patience has been very valuable to finish this work and to my little son, Ozan, who was generally patient during the times I was not available to him when completing my thesis.

Finally, I gratefully acknowledge partial financial support by Turkish Education Foundation, NSERC and URIF.

In Memory of My Beloved Father, Nurettin,

and

To Bahriye Aysel, Paul and Ozan

Acronyms

CC	Concurrency Constraint
CFSM	Communicating Finite State Machine
DBM	Directed Behavior Module
DFA	Deterministic Finite Automata
E2EC	End-to-End Constraint
EFSM	Extended Finite State Machine
ERM	Error Recovery Module
FDT	Formal Description Technique
FIFO	First In First Out
FSM	Finite State Machine
GSM	Global Scenario Machine
LC	Local Constraint
MSC	Message Sequence Chart
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PSM	Phase Scenario Machine
SAP	Service Access Point
SD	Sequence Diagram
SP	Service Primitive
TCON	Transport Connection
TDATA	Transport Data
TDISC	Transport Disconnection

Table of Contents

Chapter 1: Introduction and Motivation

1.1. Introduction and Motivation

In communications systems, the service concept is the key to a good design. One of the early papers by [Viss85] explains the use and importance of services in great detail. The communications systems are modeled as a layered architecture shown in Figure 1 on page 8 in Chapter 2. Each layer is responsible with providing a particular set of functions, called the *service*, to the layer above.

A *protocol* is a set of rules used by the protocol entities which implement the service behavior. Therefore, protocols are designed to provide services.

The process of building a protocol with a certain service or partially defined protocol specification in mind is called *protocol synthesis*. In the literature, a substantial amount of research has been published on protocol synthesis [Prob 91].

Given the complexity of protocol specifications, an important complementary topic of study has been *protocol decomposition*, which is the subject of systematically partitioning a protocol specification (usually given in a Finite

State Machine form) into modular pieces of specifications to reduce complexity. Some early studies have used decomposition as a protocol design concept, e.g. [Choi 86] and [Chow 85]. This research made valuable contributions toward achieving some desired properties for communication systems, e.g. modularity and simplicity of design [Chow 85]. In addition, it has been shown that the use of functional decomposition techniques facilitates the protocol testing process [Boyc 90].

Similar to protocol synthesis, building services from their partial or informal specifications is called *service synthesis.* However, very little work has been done on service synthesis. Some recent synthesis techniques for services are described in [Hsia 94], [Dsou 95] and [Sale 96].

The work by [Hsia 94] is one of the first good examples of defining services from informal specifications. However, only the *local view*, i.e. only one users' view of service interactions, is represented. A scenario tree is built to represent the service from a particular users' view, and the tree is converted to an FSM representation.

The recent works by [Dsou 95] and [Sale 96] have demonstrated methods to derive Finite State Machine (FSM) representations of the services from scenarios and traces, respectively. *Scenarios* are partial sequences of events which are behavior descriptions of the interactions between a system and its environment in a restricted situation [Dsou 95] and are derived from requirements or specifications of the system. Whereas, *traces* are sequences of observable

snapshots which are collected through the execution of the implementation of a system. A single trace includes information such as event observed, point of observation and timestamp. [Dsou 95] is directed toward design synthesis and [Sale 96] is directed toward design recovery from implementation.

This thesis deals with the service synthesis problem. In this thesis, a method is given for synthesizing services into global FSMs systematically and semi-automatically while keeping simplicity, completeness and modularity in mind. In the following, some research which inspired this thesis work is briefly discussed.

With respect to simplicity, a class of simple specification techniques are *constraint-oriented specification methods*, which are methods for specifying the observable behavior based on the constraints imposed on the proper temporal ordering of interactions between a system and its environment, such as in [Faci 90]. A *constraint* can be defined as a relative ordering on the occurrence of two events. A simple example from our daily life is that we must insert enough coins before we are able to make a phone call from a pay phone. A service-oriented example between two service users is: "A Connection Establishment Request message must be sent by a user before a Connection Establishment Indication message can be received by the other user". The details of these specification techniques are discussed in Chapter 2. Constraints can be used effectively when synthesizing services.

With respect to completeness, from the service user point of view, a service includes the collective behavior of network users. Due to the unpredictability of

their concurrent behaviors, message collisions can occur, and need to be represented in a service specification for completeness. This completeness requirement is also noted by [Yu 92] and [Kaku 94].

Finally, with respect to modularity, if services are synthesized into modular pieces, the derived protocol will inherently be modular. Therefore, it is desirable to derive modular service specifications [Prob 91].

In this thesis, using a similar (constraints) approach to [Chow 85], we build a Global Service Specification from a set of constraints gathered from the informal specification and Sequence Diagrams. Our goal is to be able to represent all (normal and exceptional) behaviors of the service. For example, we want to be able to recognize message collisions. Finally, our modular design approach will assists designers in designing services that are extensible.

Restriction:

The service synthesis method described in this thesis is restricted to an underlying communication medium with two FIFO channels with message capacity one between service users. Also, we consider only services where there one to one relation between a message sent at a Service Access Point (SAP) and a message received as a consequence at a remote SAP. Both of these restrictions can be removed, but most services obey our simplifying restrictions.

1.2. Contributions of the Thesis

This thesis contains the following main contributions:

- Refines the definition of service constraints; local and end-to-end, and introduces the concurrency constraints.

- Automatically constructs a Phase Scenario Machine (PSM) Construction algorithmically for each phase. PSM is an FSM which looks like a tree when being built. After it is built, each PSM contains the complete set of scenarios within a *phase* which is a sequence of message interactions intended to achieve a specific user goal. Concurrent behaviours are represented by using interleaved concurrency.

- Gives a straight forward method for coupling PSMs into a Global Scenario Machine (GSM) which is an FSM specification of the whole service.

- Extends the work by Chow et al. by introducing **collision recovery** at the phase boundaries or transition regions in the Service Specification. This allows the service specification include procedures for detecting and resolving collisions of service requests.

More details and discussion of contributions can be found in Chapter 5.

1.3. Organization of the Thesis

In the next chapter, Chapter 2, the background research leading up to this thesis work is discussed in detail. These are primarily synthesis techniques used in protocol and service design, constraint-oriented specification methods, and uses of decomposition techniques in protocol engineering.

In Chapter 3, the new approach to service synthesis is introduced. Definitions and rules to synthesize services using constraints are given as well as an algorithm to construct a Global State Machine from a given set of constraints.

Chapter 4 consists of a detailed case-study using Connection-Oriented Transport Service.

The results of this study, its limitations and a comparison to related research as

well as suggestions for future work are given in Chapter 5.

In the Appendices, a more detailed version of the case-study and the C code which implements an earlier version of the algorithm to build a Global Scenario Machine are given as well as some reference information on concurrency from [Yu 92] which is helpful to the reader.

Chapter 2: Background and Related Work

2.1. Communications Services and Protocols

This thesis assumes that the reader is familiar with the concepts of services and protocols. A good review of services and protocols can be found in [Jain 90]. The importance of services and their role in designing protocols is discussed in [Viss 85]. These concepts are briefly reviewed in this section.

> *Note: The reader who is not familiar with the standard practical notations in this area should consult the numerous references in the bibliography, especially [PTSV] and [Sari 93]. Such a reader should also consult the references on Formal Description Techniques [FORTE] and protocol synthesis techniques [Prob 91] and [Sale 91]. The most common models of services and protocols, namely Labelled Transition Systems and Extended Finite State Machines are also described in these references.*

In the OSI layered architecture (see Figure 1 on page 8), a communications system is viewed as a hierarchy of layers. Each layer entity communicates with its peer layer entity on the other end of a communication channel; for example, Protocol Entity *i* communicates with Protocol Entity *j* in Figure 1.

Figure 1 depicts the hierarchical relationship between the different layers; for simplicity only two layers are shown. Each layer (N) views its layer above, layer (N+1), as *a set of users*. The users view the layer below as a *service provider*.

Hence, the abstract behavior of each layer is called the *service*. Users can access the *service provider* through the Service Access Points (SAPs) of the service provider. The interaction between the users is realized by the service provider with exchanges of messages called *service primitives* at the SAPs.

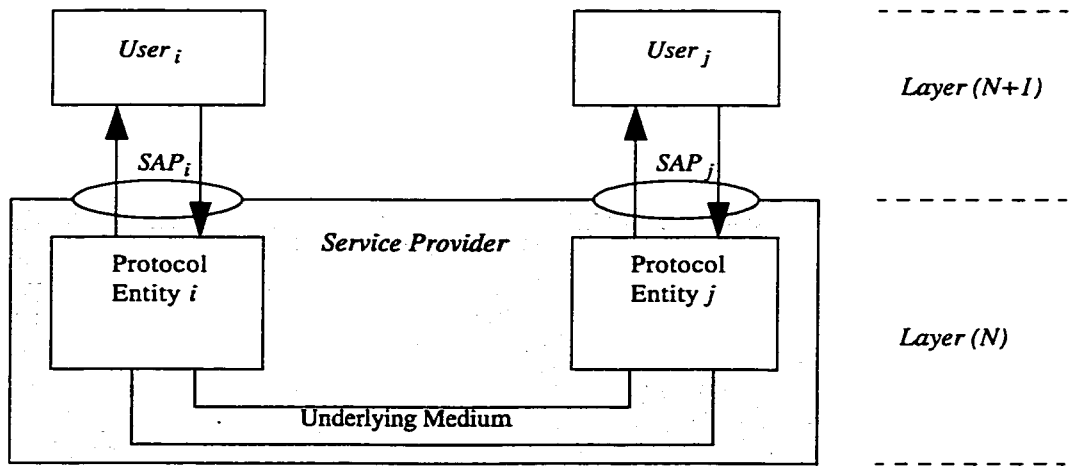


Figure 1. Layered Architecture

The communications systems are implemented by defining the services and the protocols used by the service provider to realize the services.

The protocol entities implement the service offered by the *service provider* to its users at the *SAPs*. Protocol entities exchange messages, called *Protocol Data Units(PDUs),* through the underlying service provider. The *protocol* consists of the set of rules used by each protocol entity that govern the exchange of PDUs between peer protocol entities.

Since protocols implement the service, their requirements are inherited from the

service requirements [Sale 91].

2.2. Constraint-Oriented Specifications

Protocol specification techniques have been the subject of much research. An overview of protocol specification concepts and issues can be found in [Boch 89]. This research focuses on the *constraint-oriented specification techniques*, as described in Chapter 1.

One of the earliest works on *constraints* is given by [Boch 80]. More recently, constraint-oriented specifications have been used successfully to define *LOTOS* processes [Scol 87] [Viss 88] [Viss 93] [Faci 90].

As stated in [Viss93], a *system specification* constrains the behavior of both the system and the environment. Its behavior can be defined as a possible ordering of message interactions between the system and its environment. The purpose of constraint-oriented specifications is to reflect the observable intended behavior of a system. Therefore, constraint-oriented specifications depict the externally observable behavior in terms of interactions between the system and its environment which is called the *extensional behavior* [Viss 88]. By means of constraint-oriented specification, the expected behavior of the services is limited.

Constraints can exist in the form of:

- Natural Language
- Message Sequence Charts or Sequence Diagrams
- State Machines.

In [Scol 87], [Viss 88] and [Faci 90], a widely accepted classification of the types of constraints used in LOTOS is given:

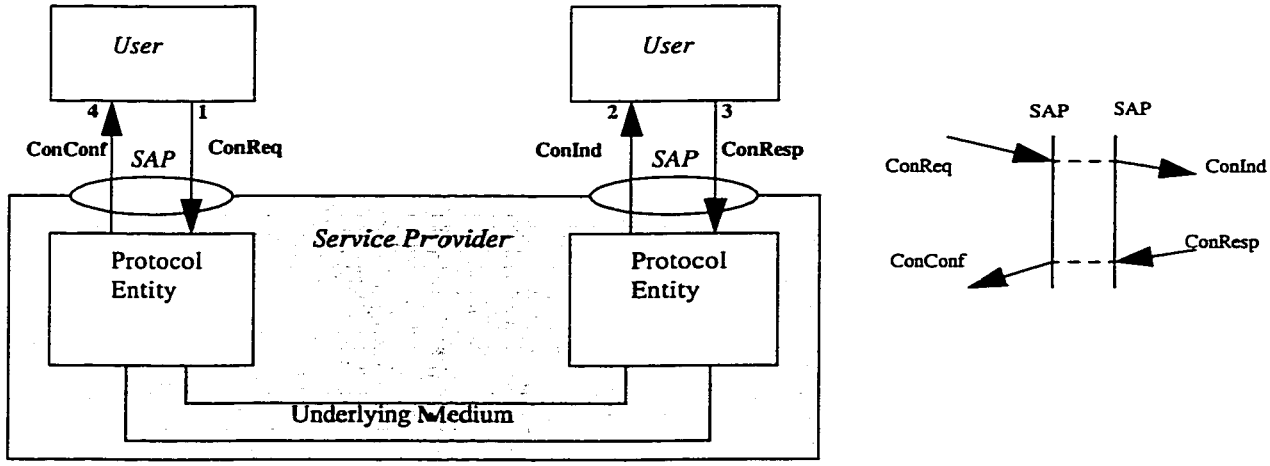


Figure 2. Interactions at SAPs

- *Remote or End-to-end constraints*: These constraints specify a partial order of the service primitives at each Service Access Point, e.g. connection request (ConReq) message causes a connection indication (ConInd) message at the other remote SAP (see Figure 2).

- *Local constraints*: These constraints specify the temporal ordering of service primitives at a single SAP. For example, before a connection indication (ConInd) message arrives , the user cannot issue a connection response (ConResp) message (see Figure 2).

- *Global constraints:* These constraints apply to values or predicates that must be satisfied system-wide. Only [Faci 90], identifies these constraints. For example, a specified maximum number of connections supported by a communications port is a global constraint.

In addition to the use of constraints in LOTOS, [ChuL 88a] describes a service model based on local and global constraints specified as FSMs. In [ChuL 88a], the global constraints are defined as *remote* or *end-to-end* constraints according to the above description.

All these works assume a synchronized communication mechanism such as in LOTOS and CSP; however an application to asynchronous communication would be interesting to implement especially in Estelle, since this type of communication is more realistic.

2.3. Protocol Synthesis Techniques

Protocol synthesis is the process of building a protocol with a certain service or partially defined protocol in mind.

A valuable review of protocol synthesis techniques is given by [Prob 91]. Mainly, protocol synthesis techniques are divided into two groups:

1. Non-service oriented or bottom-up approaches,
2. Service-oriented or top-down approaches.

In bottom-up approaches, given a partial specification of protocol entities or one complete protocol entity, the rest of the protocol is composed. These techniques

are useful because they start from a semi-formal, incomplete specification and derive the corresponding protocol entities. They are generally expensive methods since they require the extensive validation of safety and liveness properties which guarantee that the protocol specification is free from deadlocks, unspecified receptions, non-executable transitions and livelocks [Mill 90].

However, in top-down approaches, the service specification is refined into two protocol entities. These techniques start from a formally specified complete service specification and have proven to be more cost effective since they guarantee the safety and liveness properties in the process of synthesis. These techniques are able to carry the properties of services into the protocol specification through inheritance.

This research focuses only on state machine based synthesis techniques.

A recent work by [Kaku 94] has introduced a global protocol synthesis technique which uses a modified global service FSM in which the message collisions are included. The non-deterministic service FSM is modified to include message collisions that can occur due to its non-deterministic behavior. It is one of the first works to include collisions formally in the service specification.

2.4. Protocol Decomposition

Due to the complex nature of protocols, a number of decomposition techniques have been introduced. These techniques are used both in designing and testing of protocols.

2.4.1. Decomposition and Synthesis

Some bottom-up approaches have used decomposition to reduce the complexity of protocol specification. Among these techniques are [Choi 86] and [Chow 85].

[Chow 85] uses a bottom-up approach to construct protocols which are decomposed into *phases*. Informally, a *phase* is a sequence of message interactions intended to achieve a specific user goal. A formal definition of a phase is given in [Chow 85], but the identification or recognition of phases is not automated, i.e., it is left to the protocol designer.

- First, each phase is constructed individually and verified for the safety properties. The criteria used for verifying a phase is that the composite behavior of the *communicating protocol FSMs (CFSMs)* satisfies the liveness and safety properties. Also, a phase should terminate properly; i.e. the *channels* between the CFSMs should be empty at an *exit state*. The communication model is that there is an incoming and outgoing communication *channel* from a CFSM to its peer CFSM as the underlying communication medium.

- Next, each phase is connected through the exit state sets and the result is a completely specified CFSM which satisfies the safety and liveness properties. The paper presents and illustrates the method by constructing a formal specification for a protocol which is composed of more than one phase.

[Choi 86] presents a method for decomposing CFSMs into *structured partitions*.

A formal definition of a *decomposable CFSM* within this framework is given. Before an CFSM can be partitioned, it has to satisfy the decomposability rule so as to prevent *unstable* states and non-synchronizable behavior in the resulting specification. The decomposability property guarantees that there are no transitions between the structured partitions which will result in a collision (also called a "crossover"). The safety and liveness properties of the CFSMs are preserved during the partitioning process. A definition of partitions and the *transition regions* between them is given. However, a structured partition in this method does not correspond to a phase; it corresponds to any partition in the CFSM which satisfies the decomposability property. This method in [Choi 86] does not specify how the protocol subgraphs are built. The partitions are derived in a stepwise fashion until the minimum partitions which satisfy the liveness and safety properties are reached.

2.4.2. Decomposition and Testing

Decomposition is used not only in simplifying protocols for design purposes, but also in simplifying protocol specifications for testing purposes. Given a protocol specification, these methods derive a decomposed protocol specification to be used for *conformance testing* which is the black-box testing of implementation of a protocol against its specification.

[Favr 87] uses decomposition in *conformance testing* and introduces a decomposition methodology briefly. Decomposition is used to simplify the testing of Extended Finite State Machines (EFSMs), or more precisely, the

protocols represented by EFSMs in the Formal Description Technique (FDT) Estelle. The partitioning is carried out by finding the *feasible paths* for a given initial input value to the EFSM. A drawback of this kind of partitioning is that it creates redundant paths. A partitioning of the functionality is mentioned briefly involving the decomposition of the original specification as an abstract machine or EFSM into phases. Since the EFSM is often fairly large, decomposition is found to be a useful technique for generating test scenarios. A drawback is that few original specifications are given in Estelle or any other FDT.

[Boyc 89 a&b] again uses decomposition for purposes of conformance testing. The original specification is assumed to be given in Estelle. The decomposition is done by recognizing the "goals" to be performed by the protocol specification. Thus, the protocol specification is decomposed into *Directed Behavior Modules* (DBM) and *Error Recovery Modules* (ERM). DBMs correspond to goals to be achieved by the specification and each goal represents a phase. A set of definitions are given to define the transition types which can exist in a phase and a modified labeling algorithm is given to label the states so that transition types can be recognized immediately. It is a semi-automatic method. The nonexistence of certain type of transitions which are called *"regressive transitions"* in a phase is used as a criteria for decomposability, as well as the existence of other types of transitions. The resulting protocol subgraphs are used in producing test suites for conformance testing [Boyc 90].

All of the above methods are based on protocol specifications given in a state

machine form. Methods by Choi and Chow use decomposition for synthesis of protocols, Boyce and Favreau show the application of decomposition to Conformance Testing.

2.5. Service Synthesis Techniques

Service synthesis techniques are methods for composing services from their partial or informal specifications.

An early and important contribution to synthesis of service specifications is made by [Hsia 94]. In this method, interfaces are defined from a local view and represented as a set of scenarios. Then, the scenarios are validated and converted into FSMs. The local FSMs describe the behavior expected by the user according to the service provided. Non-deterministic behavior and *spontaneous transitions*, i.e. transitions caused by provider-initiated events, are not allowed in the specification. A formal grammar is defined to specify the *scenario tree*.

Recently, [Dsou 95] and [Sale 96] have provided techniques for automatically synthesizing service FSMs from scenarios and traces.

In [Dsou 95], a timed automata approach is used to synthesize services from partial scenarios. A series of *timing constraints* are used to describe the relations and temporal order of events. Although the resulting state machine is non-deterministic, the *concurrent events* which can happen in parallel are assumed to be independent of each other. In other words, it is assumed that parallel events are supported by the state machine as long as their occurrences do

not collide or interfere with each other.

In [Sale 96], a service synthesis technique from *reconciled traces* is introduced. Both the local and global constraint FSMs are derived by building a state machine incrementally. The result is a minimal FSM. However, concurrency is not handled, explicitly.

2.6. Benefits and Disadvantages of Various Design Approaches

One advantage of constraint-oriented specifications discussed in Section 2.2. on page 9 is their simplicity and practicality. Such specifications attempt to meet the requirements of their environment.

Initially, all informal specifications start with a list of requirements and expectations about the behavior of a system. Therefore, if a service satisfies the constraints imposed by its service specification, its protocol specification must inherently satisfy the constraints that are imposed by the service specification through the use of service-oriented protocol synthesis.

The decomposition techniques have been shown to achieve modularity in a complex design as discussed in Section 2.4.1. on page 13.

Usually, the only source for a service is an informal (natural language) or a semi-formal specification including state transition tables. In addition, in informal or semi-formal service specifications such as [ISO 86], the services are specified as a collection of partial sequences of interactions with accompanying text and graphics such as Sequence Diagrams or Message Sequence Charts.

Through decomposition and the application of constraint-oriented methodologies, it is possible to specify services. Consequently, protocols can be derived in a straight forward manner while achieving modularity in the specification.

In this thesis, a service synthesis method similar to the scenario-based approach in [Hsia 94] is presented. However, this approach makes use of constraints and decomposability, and generates a Global FSM. This method consists of the following steps which explained in the next chapter:

1. A list of requirements is gathered from the informal specification as a set of constraints.

2. The constraints are represented as event pairs.

3. These constraints are then used to build a Global Service Specification (FSM) semi-automatically while achieving modularity. This keeps the service specification manageable.

4. The resulting specification is able to represent and to generate the global scenarios that can be executed by the service. Since the service specification is built from its set of constraints, it also conforms to its requirements.

In the next chapter, this synthesis approach is discussed in greater detail. The formal definitions are given and the method is explained with algorithms and rules.

Chapter 3: A Service Synthesis Method From Sequence Diagrams

3.1. Introduction

In this chapter, a new approach to specifying services is introduced. First, some preliminary information about the communication and the service specification model is given followed by an overview of the new synthesis method. Then each step is explained in detail along with the formal definitions and algorithms for the synthesis method. Lastly, the use of minimization techniques for *Finite State Machines* is discussed for minimizing the service specification.

3.1.1. Communication Model

In Figure 3 on the next page, the communication model used by the algorithm is shown. The service provider is seen, through Service Access Points (SAP), by its users as a communication medium made up of FIFO channels [Boch 80, Choi 85].

The channel from user *i* to user *j* is denoted with symbol C_{ij} [Choi 85] and vice versa. The capacity of the channel is denoted by $|C_{ij}|$.

For the sake of simplicity, the capacity of a channel is assumed to be one message at a time, i.e. $|C_{ij}|=1$ and $|C_{ji}|=1$.

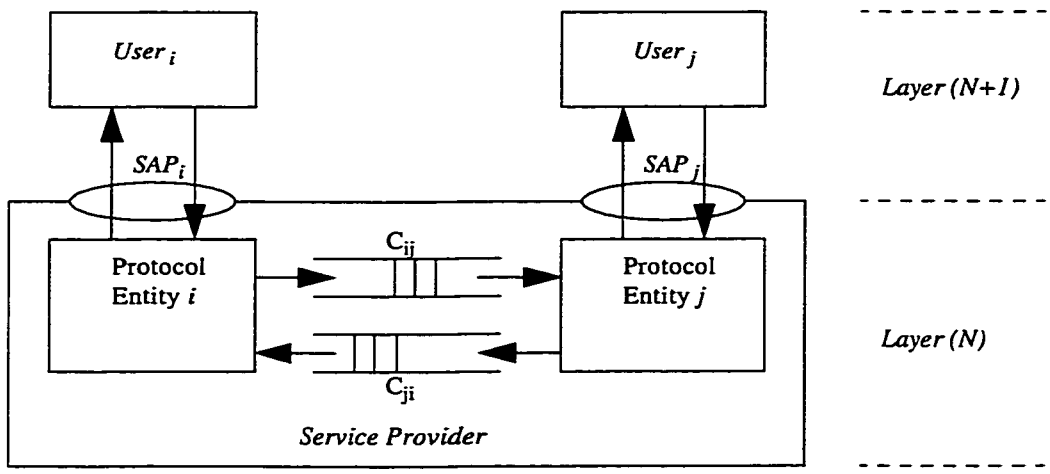


Figure 3. The Communication Model

3.1.2. Service, Service Elements and Phases

Communications systems are modeled as a layered architecture. Each layer represents a different level of abstraction in the communications system. (N)-layer, service provider, provides a set of functions, called service, through SAPs to (N+1)-layer entities, called service users. [Viss 85] discusses service concept and its importance in detail.

A service is a collection of individual services, called *service elements*, made available by a service provider to its users [Jain 90].

An example of a service such as Transport Service, and its properties are seen in Table 1 on page 21. In this example, the following is immediately is noticed:

- A service is either connection-oriented or connection-less.
- A service consists of phases associated with at least one service element.

- A service element is associated with only one phase and a set of unique service primitives.
- A service element can be mandatory or optional, confirmed or unconfirmed, user-initiated and/or provider-initiated.

Type of Service	Service Element	Nature of Service	Primitives
Connection Oriented	**Establishment Phase**		
	TC establishment	Confirmed and Mandatory	T-CONNECT request T-CONNECT indication T-CONNECT response T-CONNECT confirm
	Data Transfer Phase		
	Normal data transfer	Unconfirmed and mandatory	T-DATA request T-DATA indication
	Expedited data transfer	Unconfirmed and mandatory but user optional	T-EXPEDITED DATA request T-EXPEDITED DATA indication
	Release Phase		
	TC Release	unconfirmed or provider-nicti-tated and mandatory	T-DISCONNECT request T-DISCONNECT indication
Connection-less	**Data Transfer Phase**		
	Unit data transfer	Unconfirmed, provider and user optional	T-UNITDATA request T-UNITDATA indication

Table 1: Example specification of services provided by a layer [Jain 90]

Each phase represents a stage within the service. If users are trying to establish connection, it is said that the users are in the connection establishment phase. All that is involved (i.e. sequences of interactions) in achieving a connection establishment between the users takes place in the connection establishment phase.

Each service element is associated with a specific procedure implemented by the service provider. An example of a service element is given in Figure 4. Sometimes more than one service element may be similar in nature, i.e. achieve

the same intended goal with a slight difference. For example, expedited data transfer and normal data transfer service elements accomplish the same goal data transfer. Then, these service elements fall into the same phase category, i.e. data transfer phase. A more in-depth information about services and service elements can be found in [Jain 90].

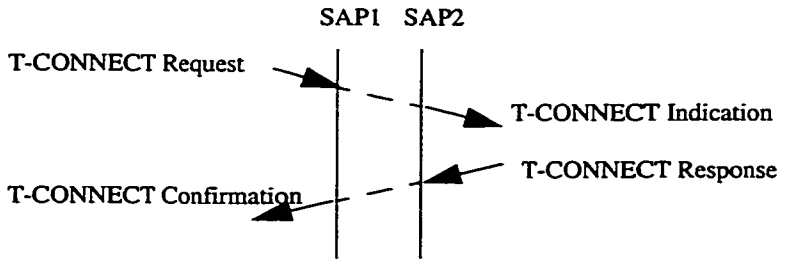


Figure 4.T-CONNECT service element in Connection-Oriented Transport Service

A *phase* is a sequence of message interactions intended to achieve a specific user goal such as data transfer and releasing of an existing connection [Boyc 90]. A **phase** is mapped to a one or more service elements of the same nature and their associated set of unique Service Primitives (SPs) in a service specification. Most connection-oriented communications services consist of three phases: connection establishment, data transfer and connection release.

3.1.3. Formal Service Specification Model and Basic Definitions

Some notations and definitions are introduced here. They will be used to describe the synthesis method in the following sections.

In our service synthesis method, first all of the phases are specified and then the phases are coupled to form the complete service specification. Therefor, we refer

to two kinds of specification; namely PSM and GSM:

- *Phase Scenario Machine (PSM)*; This specifies the possible interactions during only one phase in a given service. All the possible scenarios which can be observed globally (i.e. distributed over SAPs) of a phase are specified by its PSM.

  A PSM is a global, deterministic *Communicating Finite State Machine (CFSM)* which is a directed graph made of vertices (states) and edges (transitions). A PSM is built iteratively by adding events which is discussed in Section 3.4. on page 40.

- *Global Scenario Machine (GSM)*; This is the complete service specification. All of the possible global (i.e. distributed over SAPs) scenarios observed in a service are specified by its GSM.

  Like a PSM, a GSM is a global, deterministic CFSM. After all of the PSMs are specified, they are coupled to form the GSM. Any two phases are coupled if there are any events which connect them to each other as discussed in Section 3.5. on page 57.

Common Concepts and Definitions. These apply to both GSM and PSM.

In a CFSM communication model, two users communicate over FIFO channels (one in each direction). A CFSM consists of vertices which represent states and edges which represent transitions labelled with a sending or receiving event. For more details, please see [Chow 85], [Mill 90].

In our specification model, events have two attributes; an *SP (message)* attribute and a *SAP (location)* attribute. Receiving and sending actions on events are indicated by (-) for a *send event* and (+) for a *receive event*. For example, -TCONReq@SAP_1 is a send event, see Figure 4 on page 22.

- An **event** has two attributes; an SP and a SAP. An event with a (+) sign indicates a *receive event* and an event with a (-) sign indicates *send event*. For example, +TCONInd@SAP_2 is a receive event where service primitive TCONInd is received at service access point 2.

The concept of *send* and *receive event* is used in defining the constraints and through out the synthesis method both in PSM construction and coupling of PSMs. Therefore, it is important to label each event as either a receive or a send event.

Our model is a global CFSM which represents the behavior of the service distributed over all of its SAPs. It is also a deterministic CFSM; all outgoing transitions of a state have different labels [Linz 90]. We can formally specify our model as the following;

- A global deterministic CFSM is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where,

  Q: is a finite set of states.

  Σ: is an alphabet of events; $\Sigma = \{e | e = \pm SP_i @ SAP_j\}$ where

  "+" means "*receive*",

  "-" means "*send*",

  $(1 \le i \le n)$ (n is the number of service primitives) and SP is the set of service primitives,

  $(1 \le j \le m)$ (m is the number of service access points) and SAP is the set of service access points (accessible by the users).

  δ: is the next state function; $\delta(q_i, e) = q_j$ where $e \in \Sigma$.

  q_0: is the initial state.

  F: is a set of final state(s).

- In addition, the following function is defined to extract the SAP attribute of a given event for convenience:

  SAP(e)=SAP_j where $e \in \Sigma$. This function returns the SAP attribute of a given event.

Up to this point, we defined the common concepts used by both PSM and GSM. Due to their synthesis process, PSM and GSM also needs to use different concepts. In the following, we discuss first the concepts used only by a PSM and then the concepts used only by a GSM.

Phase Scenario Machine. A PSM of a phase is constructed incrementally by adding events starting with an initial event from the set of events which belong the phase (how these events are found and added will be discussed in later sections of this chapter). Therefor, a PSM looks like a tree when being constructed. Thus, operations used on trees can also be used on a PSM being constructed. However, after this incremental construction is completed, a PSM will be transformed into a CFSM and no longer can the tree operations be applied.

A **final state** marks the end of a completed PSM. At a final state of a phase, the channels between the two users are empty in both directions. In phases which can repeat themselves (cyclic phases), the final state is also the initial state. Otherwise, the final state has no outgoing edges. A phase can have more than one final state.

During and after the construction of a PSM, there is only one transition at most between any two states in the PSM. Therefor, a transition is defined as follows:

- A **transition** is represented as $t_{ij} = q_i \xrightarrow{e} q_j$, where $\delta(q_i,e) = q_j$ and the **label** of transition t_{ij} is $l(t_{ij}) = e$.

Earlier, it was mentioned that a PSM being constructed looks like a tree. The following definitions apply only to a PSM being constructed. These definition are important for the PSM construction algorithm described in Section 3.4. on page 40.

- A **subtree** in *PSM* is denoted by T_i, where q_i is the root of the subtree.

- A **path** p_{in} is a series of transitions from q_i to q_n; $p_{in} = t_{ij} \cdot t_{jk} \cdot \ldots \cdot t_{mn}$ where "." is the concatenation operator.

  P_i is the set of all possible paths p_{in} from q_i to q_n where q_n is a leaf node.

  For simplicity, a subset of paths of P_i such that $(t_{ij}.P_j)$ is denoted by P_i^j. This notation is needed when we refer to a set of specific paths leading from a certain transition. p_{in}^j denotes a path starting from q_i with the transition t_{ij} and to the final state q_n.

  The tree shown in Figure 5 on page 26 helps illustrate this notation. Symbolic names are used to relate the actual notation.

  T_1 is a tree with the initial state q_1 and the leaf nodes are q_4, q_7 and q_9.

  P_1 is the set of all possible paths that lead to q_4, q_7 and q_9.

  $P_1 = \{(t_{12}.t_{23}.t_{34}), (t_{15}.t_{56}.t_{67}), (t_{15}.t_{58}.t_{89})\}$

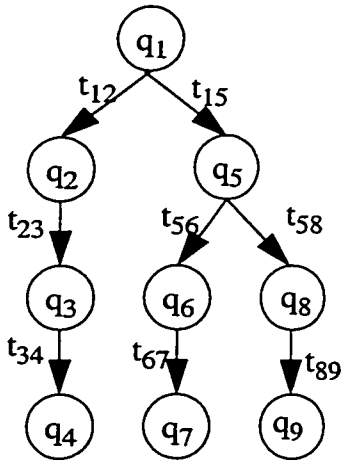


Figure 5.A subtree T_1

The paths that start with transition t_{15} are denoted with P_1^5 :

$P_1^5 = t_{15}.P_5$ or $P_1^5 = \{(t_{15}.t_{56}.t_{67}), (t_{15}.t_{58}.t_{89})\}$.

Then, $p_{17}^5 = (t_{15}.t_{56}.t_{67})$ and $p_{19}^5 = (t_{15}.t_{58}.t_{89})$

These concepts are used by the PSM construction algorithm (please see Section 3.4. on page 40).

Global Scenario Machine. After all the PSMs are built, they are coupled to form the complete global service specification called GSM. The following properties apply only to a GSM:

- There can be more than one **transition** between two states in a GSM.
- Since a service usually has the capability to restart (i.e. return to its initial state), a **final state** is usually the initial state and there is only one final state in its GSM.

3.1.4. Constraints

Constraints describe the allowed order of occurrence of two events with respect to each other. Constraints are used to specify the order of events within a phase, and between phases. A constraint is simply represented by an event pair, e.g. (-TCONReq@SAP_1, +TCONInd@SAP_2). The relationship between the event pair is identified by the type of constraint set it belongs to. These constraints are derived from Sequence Diagrams (SDs - also known in practice as Message Sequence Charts (MSCs) or simply interaction scenarios). How the different types of constraints are obtained is discussed in Section 3.3.1. on page 37 and illustrated in Figure 8 on page 39.

For shorter reference to an event pair which is an element of a certain constraint set, the terms event pair and constraint pair is used interchangeably in the rest of this thesis.

In the following, the local and end-to-end constraints are defined along with the concurrency constraints.

a) *Local constraints:* These dictate the proper temporal ordering of service

primitives at a given SAP as discussed on page 10 in Chapter 2. This is the same definition as in [Boch 80, Faci 91].

A *local constraint* is an event pair which corresponds to an allowed sequence of two events of which the first event is followed by the second event and whose SAP attributes are identical.

For example, event pair (-TCONReq@SAP_1, +TCONConf@SAP_1) is a *local constraint.* It implies that -TCONReq@SAP_1 can be followed by +TCONConf@SAP_1 (see Figure 4 and Figure 8-a) and both events have the same SAP attribute. Thus, this constraint pair dictates that first TCONReq is sent at SAP_1, then TCONConf can be received at SAP_1. All such event pairs make up the set of Local Constraints *LC*.

- Formally, a set of *Local Constraints (LC)* is:

$LC = \{ (x,y) |$ where $x \in \Sigma$ and $y \in \Sigma$, and x can be followed by y and $SAP(x) = SAP(y) \}$.

b) *End-to-end constraints:* This type of constraint maps a send event at a SAP to a receive event at a different SAP. Note that this is a more refined definition of end-to-end constraints than the works discussed in [Boch 80, Faci 91].

Assumption 1: It is assumed that for each send event only one receive event can appear in the service and vice versa. In other words, a send event is mapped to a unique receive event in the service, i.e. there is a 1-1 mapping between an event sent at a SAP and an event received as a consequence at a remote SAP.

The above assumption simplifies the synthesis method.

An *end-to-end constraint* is an event pair which corresponds to a valid sequence where the first event triggers the second event and the SAP attributes of both events are different.

For example, event pair (-TCONReq@SAP_1, +TCONInd@SAP_2) is an *end-to-end constraint*. Both events have a different SAP attribute. The first event -TCONReq@SAP_1 in the event pair is a send event and the second event +TCONInd@SAP_2 is a receive event which is the direct cause of -TCONReq@SAP_1 (see Figure 4 and Figure 8-b). In other words, when TCONReq is sent at SAP_1, it stimulates a TCONInd to be received at SAP_2. All such event pairs make up the set of End-to-End Constraints *E2EC*.

A set of End-to-End Constraints (*E2EC*) is defined as:

$E2EC = \{ (x,y) |$ where $x \in \Sigma$ and $y \in \Sigma$, and $-x$ is a stimulus for $+y$ (x is a send event, y is a receive event), and $SAP(x) \neq SAP(y)$ $\}$.

c) *Concurrency constraints:* This type of constraint defines the service behavior that can concurrently be executed without causing a loss of synchronization in the behavior of both parties involved in the interaction. These constraints are used to describe the allowable collisions within a phase.

A *concurrency constraint* is an event pair where it is valid for both events to be either received or sent simultaneously at different SAPs. In other words, the two receive or send events are considered to be concurrent at different SAPs if either event occur "almost" at the same time, i.e. a collision occurs. Concurrency constraints are a new contribution of this thesis work.

More precisely, the concurrent events can be explained with *relative clocks* concept introduced by [Yu 92]. The implementation rules for relative clocks

can be found in Appendix C.

For example in Figure 8-c on page 39, both parties may send a TDATAReq or receive a TDATAInd at the same relative time [Prob 92]. The concept of *relative clocks,* proposed by [Yu 92], is a method of reconciling traces collected from distributed SAPs. The traces are reconciled based on their relative timestamps. If two events are occurring at the same *relative time*, they are *relatively concurrent* [Yu 92]. Informally, two events occurring at different SAPs are said to be relatively concurrent if either event could possibly occur before the other. Such events are of special interest for design.

Figure 8-c depicts this kind of situation where events -TDATAReq@SAP_1 and -TDATAReq@SAP_2 are relatively concurrent. A more informal explanation is that -TDATAReq@SAP_1 and -TDATAReq@SAP_2 occur roughly at the same real-time (at the exact relative-time) whereby no other event on either SAP executes until -TDATAReq@SAP_1 and -TDATAReq@SAP_2. The same is also said for +TDATAInd@SAP_1 and +TDATAInd@SAP_2. Consequently, event pairs (-TDATAReq@SAP_1, -TDATAReq@SAP_2) and (+TDATAInd@SAP_1, +TDATAInd@SAP_2) are concurrency constraint pairs.

CC={ (x,y)| x and y can concurrently happen within the same relative time, both x and y are either receive or send events, and $SAP(x) \neq SAP(y)$, where $x \in \Sigma$ and $y \in \Sigma$ }.

Examples of *Local, End-to-End* and *Concurrency Constraint Sets* can be found on page 71 in Chapter 4.

The following are some concepts used during the construction of the Phase

Scenario Machines. They are included here since these concepts are derived from the above definitions of the constraint sets.

Given a transition t_{ij} and x, the label of t_{ij}, $x = l(t_{ij})$, where $x \in \Sigma$ and $y \in \Sigma$; the following sets are defined for event x:

- $LC_x = \{(x, y) | (x, y) \in LC\}$ is the set of all local constraints whose first event is x in the set of LC. LC_x is a subset of LC.

  Example: Given (-TCONReq@SAP$_1$,+TCONConf@SAP$_1$) $\in LC$, the following can be written
  (-TCONReq@SAP$_1$,+TCONConf@SAP$_1$) $\in LC_{\text{-TCONReq@SAP1}}$.

  For convenience, it will be said that $LC_x = LC_{l(t_{ij})} = LC_{t_{ij}}$ when referring to the label of a specific transition in the PSM construction algorithm.

  Similarly;

  $LC_{x^{-1}} = \{(y, x) | (y, x) \in LC\}$ is the set of all local constraints whose second event is x in the set of LC. $LC_{x^{-1}}$ is a subset of LC.

  Example: Given (-TCONReq@SAP$_1$,+TCONConf@SAP$_1$) $\in LC$, the following can also be written
  (-TCONReq@SAP$_1$,+TCONConf@SAP$_1$) $\in LC_{\text{+TCONConf@SAP1}^{-1}}$.

  It will be said that $LC_{x^{-1}} = LC_{\Gamma^{-1}(t_{ij})} = LC_{t_{ij}^{-1}}$ when referring to the label of a specific transition in the PSM construction algorithm.

- $E2EC_x = \{(x, y) | (x, y) \in E2EC\}$ is the set of all end-to-end constraints whose first event is x in $E2EC$. $E2EC_x$ is a subset of $E2EC$.

  Example: Given (-TCONReq@SAP$_1$,+TCONInd@SAP$_2$) $\in E2EC$, the following can be written

$(\text{-TCONReq@SAP}_1,\text{+TCONInd@SAP}_2) \in E2EC_{\text{-TCONReq@SAP1}}$.

It will be said that $E2EC_x = E2EC_{l(t_{ij})} = E2EC_{t_{ij}}$, when referring to the label of a specific transition in the PSM construction algorithm.

Similarly;

$E2EC_{x^{-1}} = \{(y,x)|(y,x) \in E2EC\}$ is the set of all end-to-end constraints whose second event is x. $E2EC_{x^{-1}}$ is a subset of $E2EC$.

Example: Given $(\text{-TCONReq@SAP}_1,\text{+TCONInd@SAP}_2) \in E2EC$, the following can also be written

$(\text{-TCONReq@SAP}_1,\text{+TCONInd@SAP}_2) \in E2EC_{\text{+TCONInd@SAP2}^{-1}}$.

It will be said that $E2EC_{x^{-1}} = E2EC_{\Gamma^{-1}(t_{ij})} = E2EC_{t_{ij}^{-1}}$ when referring to the label of a specific transition in the PSM construction algorithm.

- $CC_x = \{(x,y)|(x,y) \in CC\}$ is the set of all concurrency constraints whose first event is x. CC_x is a subset of CC.

Example: Given $(\text{-TDATAReq@SAP}_1,\text{-TDATAReq@SAP}_2) \in CC$, the following can be written

$(\text{-TDATAReq@SAP}_1,\text{-TDATAReq@SAP}_2) \in CC_{\text{-TDATAReq@SAP1}}$.

It will be said that $CC_x = CC_{l(t_{ij})} = CC_{t_{ij}}$ when referring to the label of a specific transition in the PSM construction algorithm.

Similarly;

$CC_{x^{-1}} = \{(y,x)|(y,x) \in CC\}$ is the set of concurrency constraints whose second event is x. $CC_{x^{-1}}$ is a subset of CC.

Example: Given $(\text{-TDATAReq@SAP}_1,\text{-TDATAReq@SAP}_2) \in CC$, the following can also be written

$(\text{-TDATAReq@SAP}_1,\text{–TDATAReq@SAP}_2) \in CC_{\text{–TDATAReq@SAP2}^{-1}}$.

It will be said that $CC_{x^{-1}} = CC_{\Gamma^{-1}(t_{ij})} = CC_{t_{ij}^{-1}}$ when referring to the label of a specific transition in the PSM construction algorithm.

Detailed examples of the use of these concepts are given in Chapter 4.

3.1.5. Overview of the New Synthesis Method

The new synthesis method uses a phase-based approach similar to that described in [Chow 85]. However, the new service synthesis algorithm is developed to construct the Global Scenario Machine of a service from its informal specification based on a given set of constraints.

The overall method for global service synthesis from sequence diagrams is shown in Figure 6 on page 34. Steps 1 and 2 are the preparation stages for input to the synthesis method. They are the guidelines for the designers to collect the necessary input for the synthesis process. If all the inputs are readily available, these steps need not be executed. But they can be used for checking the completeness of inputs. Steps 3 and 4 make up the new approach to synthesizing global service. Step 5 reduces the service specification to a minimal GSM.

In step 1, the informal specification is analyzed for consistency. This involves verifying whether the text has more interaction information between the service users than indicated by the Sequence Diagrams (SD) included in the specification, if any. In our approach, it is assumed that at least some SDs are included in the informal specification. They can be completed by checking against the rest of the informal specification, e.g. text, tables, etc.

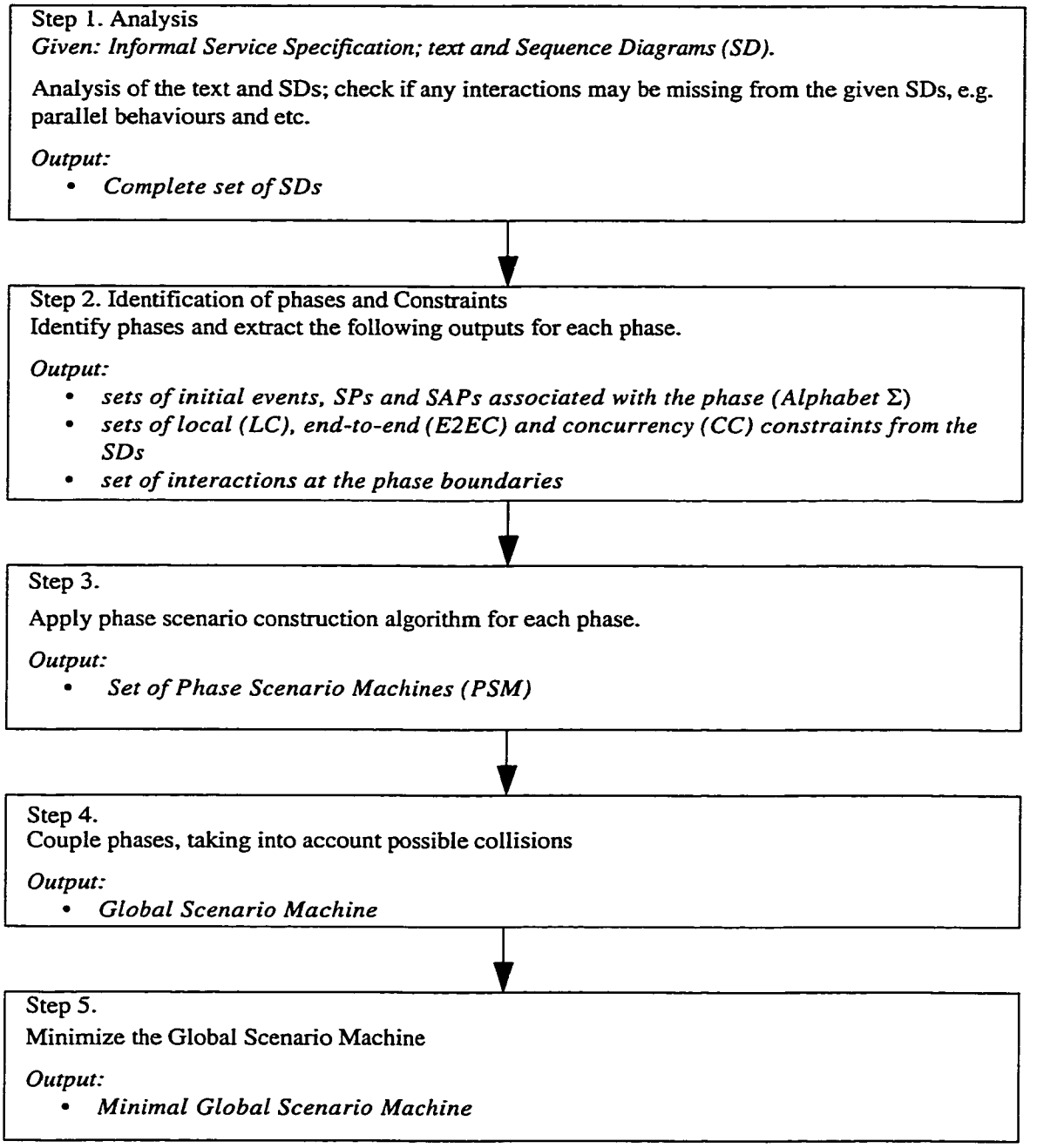


Figure 6.The Service Synthesis Method from Sequence Diagrams

In step 2, all the key information about the phases of the service and the associated attributes of phases are identified from the informal specification.

These key attributes are initial events, service primitives and their associated Service Access Points, local constraints, end-to-end constraints, concurrency constraints, and boundaries or transition regions [Chow 85] between phases and their interactions.

In step 3, given a set of outputs from step 2, the constraint-oriented algorithm constructs a set of *Phase Scenario Machines (PSMs)*. Each Phase Scenario Machine is a deterministic finite automaton applicable to one phase of the service. As a result, a set of *Phase Scenario Machines (PSMs)* is obtained.

In step 4, all *PSM*s constructed in step 3 are coupled at their phase boundaries to form the composed *Global Scenario Machine (GSM)*. This step uses the same principles as [Chow 85] and [Choi 86]. We extend their methods so that possible collisions at the phase boundaries are also identified and included in the GSM of the service.

From step 4, the resulting GSM may not be a minimal FSM. Therefore, in step 5, existing FSM minimization methods such as the one described in [Koha 78] or [Linz 90] can be used to obtain the minimal Global Scenario Machine representation of a service.

3.2. Step 1: Analysis of the Informal Specification

In an informal specification, the behavior of the service provider is usually described as a set of partially defined scenarios. These are in the form of Sequence Diagrams (SD) or Message Sequence Charts (MSC) along with the text,

such as in [ISO 86]. Our method could support both SD and MSC. However, SDs are chosen here to describe the synthesis method. Note that the set of SDs may not always be complete and some important additional details may be hidden and only implied in the English text of the specification.

Given an informal specification such as [ISO 86], the very first step is to verify all the SDs against all the information, e.g. text, tables and partial FSMs in the informal specification. The SDs are completed by adding any interactions between the service users, which are specified in the text or diagrams but missing from the current set of SDs, if any. This is important because the algorithm assumes that a complete set of SDs exists. Otherwise, the constraints of the service will not be complete and the resulting GSM will be incomplete.

3.3. Step 2: Identification of Phases and Constraints

When a service is specified, its phases are naturally identified even in its informal specification, e.g. [ISO 86].

The attributes of a phase are its set of service primitives, their associated service access points, the proper sequence(s) of these service primitives and the initial service primitives at the service access points that start the phase.

Since the synthesis method generates a global GSM, an event always has two attributes: a service primitive and the service access point at which it can occur. In the algorithm, all the events are referred to in this context.

As discussed in Chapter 2, in works by [Chow 85] and [Choi 86], the transition

regions between decomposed protocol partitions are identified and their transition criteria are specified. In [Chow 85], transition regions are between phases and the transitions made are determined by a set of *exit state pairs*. In [Choi 86], a similar definition is made for any possible structured partition in the protocol specification. Both methods use Communicating Finite State Machines as a specification model.

We use the term **phase boundary** for the transition region between two phases. An example of an interaction at the phase boundary from Transport Connection Establishment phase to Transport Connection Release phase in the connection-oriented Transport Service is shown in Figure 7. In this step, the phase boundaries and their relevant interactions are identified as well as the phases themselves.

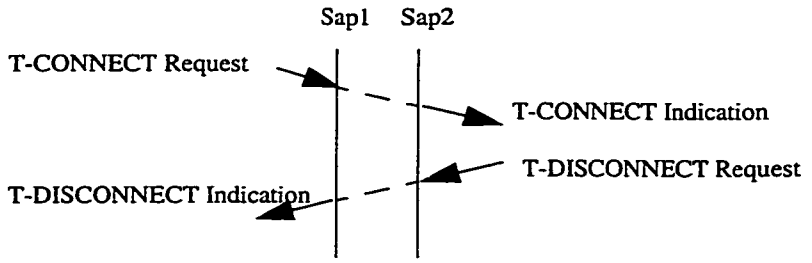


Figure 7. Transition from T-CONNECT phase to T-DISCONNECT phase

3.3.1. Obtaining Constraints from SDs

The synthesis algorithm builds the individual phases using the types of constraints defined in Section 3.1.4. on page 27.

The constraints on the service users can be obtained from the informal

specification if they are not explicitly given. Specifically, the constraints can be extracted easily as pairs of interactions in a semi-automated way if there is a complete set of SDs which describe the user interactions for each phase. This is described below:

- *Local constraints* can be extracted by slicing the SD vertically between the different SAPs, see Figure 8(a). Each slice captures the local constraint of a particular user. This yields a set of local constraints (*LC)* for each phase, whose elements are event pairs.

  For example, in Figure 8-a, it is shown how the following pairs of local constraints can be obtained from a SD:

  LC= { (-TCONReq@SAP_1, +TCONConf@SAP_1),

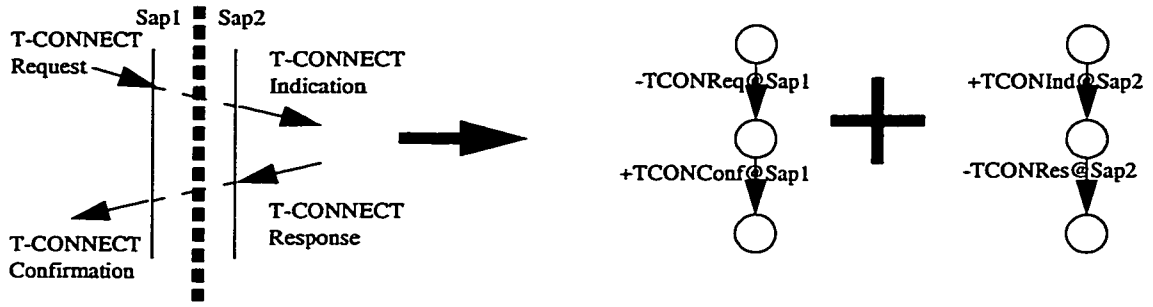
  (+TCONInd@SAP_2, -TCONRes@SAP_2)}

- *End-to-end constraints* (*E2EC*) can be extracted by slicing the SDs horizontally between the pairs of one input and one output from a SAP to different SAP, (see Figure 8(b)). For each phase, this yields a set of end-to-end constraints *E2EC* whose elements are event pairs.

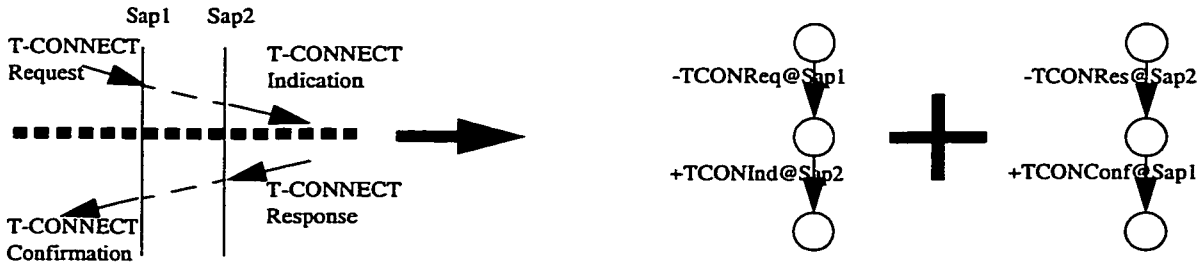
  In the example given in Figure 8(b), it is shown how the following two pairs of end-to-end constraints are obtained from a SD:

  E2EC= { (-TCONReq@SAP_1, +TCONInd@SAP_2),

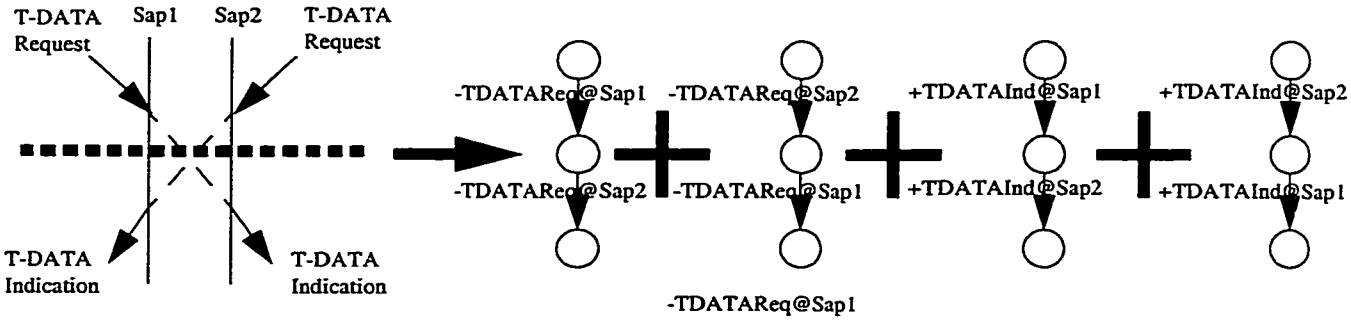
  (-TCONRes@SAP_2, +TCONConf@SAP_1)}



(a) Obtaining Local constraints through vertical slicing of SD and two pairs of interactions



(b) Obtaining end-to-end constraints in a non-colliding scenario through horizontal slicing of SD and two pairs of interactions



(c) Obtaining concurrency constraints in a colliding scenario through horizontal slicing of SD and four pairs of interactions

Figure 8. Extracting constraints from SDs

- *Concurrency constraints (CC)* are extracted from the SDs by slicing the SDs horizontally between two received or two send events on different SAPs, see Figure 8(c). These constraints reveal the allowed collisions within a phase.

Note that since the concurrent behavior is difficult to represent in a global,

deterministic CFSM, the concurrent events are shown by interleaving them.

In Figure 8(c), an example is given for obtaining the following pairs of concurrency constraints from SDs:

CC= { (-TDATAReq@SAP_1, -TDATAReq@SAP_2),
(-TDATAReq@SAP_2, -TDATAReq@SAP_1),
(+TDATAInd@SAP_1, +TDATAInd@SAP_2),
(+TDATAInd@SAP_2, +TDATAInd@SAP_1)}

3.4. Step 3: Phase Scenario Machine (PSM) Construction

In this step, a synthesis algorithm is applied to each phase with the data obtained in the previous steps. The result is a Global Scenario Machine for each phase known as PSM.

The PSM is built by adding transitions that comply with the constraints on the service. When the PSM is being built, it looks like a tree because any repeating path (scenario) found is connected to the initial state only after all the paths are generated. Therefore, it can be treated as a tree and tree operations can be used to synthesize the PSM. Once, the PSM of a phase is completed, if traversed, it will yield all the scenarios that can be handled for that phase by the service provider.

The output from this step is a set of PSMs that specify each phase implemented in the service provider.

In the following section, we give a detailed description of the algorithm along with some definitions to help build a PSM and rules for adding a new transition to the PSM.

3.4.1. Overview of the Algorithm for Synthesizing the Phase Scenario Machine

First, the terms, definitions and assumptions used in the algorithm are given below, in addition to the definitions provided in Section 3.1.3. on page 22 and Section 3.1.4. on page 27.

The following assumption is made by the algorithm:

> *Assumption 2:* It is assumed that a phase only has simple cycles, i.e. a transition can only cycle back to the initial state (of the phase) [Linz 90].

Determining a "repeated state" in PSM Synthesis Algorithm. A state is called a *"repeated state"* if the label (event) of its incoming transition exists twice on the path from the initial state to itself. The PSM construction algorithm generates all the possible scenarios for an event, given the constraints. When adding a new event (as a result, a new transition and a new state) following a path in the PSM during its construction, the new state is marked as a *"repeated state"* if an event identical to the new event was previously encountered on the path between the initial state and the new state. For example in Figure 9 on page 42, *state 3* is marked as a "repeated state". The algorithm will generate all the possible scenarios for the first of the two identical events. Doing the same for the second (the new event) of the two identical events would result in the same or a subset of scenarios found from the first time.Therefore, no more new paths need to be generated following a *"repeated state"*. The *"repeated state"* concept is used by the algorithm as a stopping criteria for generating scenarios. This is defined formally below.

Given a path p_{ik} and a new event *e*, ***e* will lead to a "repeated state"**;

if $\exists t_{xy}$ in $p_{ik}|\ \ e = l(t_{xy})$.

e is said to lead to a "repeated state" when it is added to path p_{ik} if there is a transition already in p_{ik} whose label is also e.

Definition 1: Let p_{in} be a path and t_{mn} be the last transition in p_{in};
q_n is said to be a **"repeated state"** and t_{mn} is said to **lead to a "repeated state"**;

if $\exists t_{xy}$ in $p_{in}|\ l(t_{xy}) = l(t_{mn})$ where $t_{xy} \neq t_{mn}$ and $i \leq x < y \leq m < n$

then q_n is a *"repeated state"*

The repeated states need to be identified in a *PSM* so that the scenario generation will not re-explore the loops back into a previous path.

After the PSM of a phase is completely built, for any given transition t_{xy}, q_x is considered the same as the initial state q_0 if q_y is a "repeated state", based on Assumption 2 on page 41. The synthesis algorithm explores all the scenarios for a given event. Note that every phase has its own set of initial events (as well as one initial state) and that q_x is mapped to q_0 of its phase because its outgoing transition will only lead to a subset of the scenarios which will be generated in paths preceding q_x. There is no use in re-exploring cyclic paths.

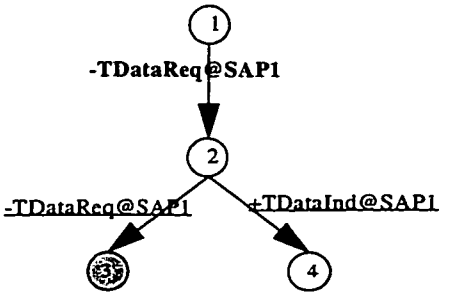


Figure 9. Example of a "repeated state"

Example: The example in Figure 9 is taken from Chapter 4. In this example, state 3 is marked as a "repeated state" because the label of its incoming transition already exist in the path from state 1 to 2.

Selecting a legal path compatible with a new event *e*. Given a new event *e*, a *legal path* for this new event is a path such that for each transition *t* in this path, constraint pair $(l(t),e)$ is a member in one of the constraint sets, *LC*, *E2EC* or *CC* where $l(t)$ denotes the label of transition *t*. The legal path ensures that a new event can be attached to a path after every transition. This is discussed in Section 3.4.4. Propagating an Event into a Path on page 50.

Definition 2: Let t_{xy} be a transition in p_{ik} and *e* be an event to be added to p_{ik}. A path p_{ik} is said to be a legal path for *e*;

if $\forall t_{xy}$ in $p_{ik} | (l(t_{xy}), e) \in (LC_{t_{xy}} \cup E2EC_{t_{xy}} \cup CC_{t_{xy}})$ where $i \le x < y \le k$

then p_{ik} is a *legal path* for event *e*.

This definition can also be explained as a simple algorithm. Note that we simplified the notation used for transitions and paths in the following for ease of understanding:

Given a path $p_n=(t_1 \cdot t_2 \cdot ... \cdot t_n)$ and a new event *e*;

```
for i=1 to n do
      l = label(t_i)
       if ((l, e) is in LC_l or E2EC_l or CC_l)
       then continue
       else stop 'the legal path is p_{i-1}=(t_1 · t_2 · ... · t_{i-1})'
end for
```

Example: Given a path $p=(-TDATAReq@SAP_1 \cdot +TDATAInd@SAP_1)$ (see Figure 9) and a new event $+TDATAInd@SAP_2$, path p is a legal path for $TDATAInd@SAP_2$ because $(-TDATAReq@SAP_1, +TDATAInd@SAP_2)$ is a

member in set E2EC and (+TDATAInd@SAP_1, +TDATAInd@SAP_2) is a member in set CC (see the sets of constraints for Data Transfer Phase on page 71 in Chapter 4).

Determining a legal path following the current transition t_{ij}: Let t_{ij} be the current transition, p_{in} be the current path from q_i to q_n where q_n is the last state in the path and e be an event to be added to p_{in}. A legal path is searched from q_i to q_m where $m \leq n$.

Determining a legal path preceding the current transition t_{ij}: Let t_{ij} be the current transition, p_{0i} be a path from q_0 to q_i and e be the current event to be added to p_{0i}. A legal path p_{lm} is searched backwards (in reverse order) from q_i to q_o where $i \geq m > l > 0$.

The PSM construction algorithm. For every phase of the service specification, a Phase Scenario Machine is built using the phase construction algorithm. The algorithm starts from the initial state (of the phase) and builds the PSM in a tree-like manner. For a detailed example, see Section 4.3. on page 70.

The following algorithm is executed for each phase. Note that the algorithm is explained in a top-down manner.

Input: initial events, set of events, *LC, E2EC* and *CC* of each phase.

Output: a PSM specification of each phase.

Algorithm for PSM Construction:

1. For all initial events;
   1.1. Add an initial transition corresponding to an initial event to initial state q_0 of the *PSM*
   1.2. For every transition t_{ij} in *PSM* where $0 \le i < n$ and $i < j \le n$ (n is the current number of states) and q_i is not a "repeated state".
      1.2.1. Build all the possible paths leading from t_{ij} by adding transitions which conform to $LC_{t_{ij}}$, $E2EC_{t_{ij}}$ or $E2EC_{t_{ij}^{-1}}$, and $CC_{t_{ij}}$ respectively:
         1.2.1.1. a) Add transitions whose labels correspond to a second event in the set of local constraint pairs of t_{ij} namely $LC_{t_{ij}}$.
            b) Merge identical paths.
            c) If any repeated states are created preceding another "repeated state" previously identified, the last "repeated state" is removed.
         1.2.1.2. a) Add transitions whose labels correspond to an event in the set of end-to-end constraint pairs of t_{ij} namely $E2EC_{t_{ij}}$ and $E2EC_{t_{ij}^{-1}}$.
            b) Merge identical paths.
            c) If any repeated states are created preceding another "repeated state" previously identified, the last "repeated state" is removed.
         1.2.1.3. a) Add transitions which conform to the concurrency constraint pairs of t_{ij} namely $CC_{t_{ij}}$.
            b) Merge identical paths.
            c) If any repeated states are created preceding another "repeated state" previously identified, the last "repeated state" is removed.

2. Remove all the repeating transitions from each path in $PSM \rightarrow PSM'$
3. Label each state in PSM'

3.4.2. Explanation of the Algorithm

The algorithm builds all of the possible paths leading from an initial event, recursively. The nodes are traversed with the pre-order tree traversal method [Aho 74].

The algorithm is explained below. However, it is suggested to refer to Section 3.4.3. on page 50 for conserving the local constraints and Section 3.4.4. on page 50 for propagation rules.

Each transition is evaluated against the local, end-to-end, concurrency constraints of its event. After a series of conditions are met, a new event is added to the PSM if it is a member of a constraint pair in the constraint sets of the current transition being evaluated.

Below, note that the current transition being evaluated is denoted by t_{ij} and its event is denoted by x. An event y denotes a member of a constraint pair whose other member is x.

- *Adding events that correspond to local constraint pairs of the current event*

1.2.1.1.a) Add transitions whose labels correspond to a second event in the set of local constraint pairs of t_{ij} namely $LC_{t_{ij}}$.

The details of this step from the algorithm on page 45 is as follows:

A) Find all y in Σ such that $(x, y) \in LC_x$ where $x = l(t_{ij})$.

B) For all p_{in}^j in P_i^j.

B.1) if there is no t_{lm} such that $\exists y = l(t_{lm})$ in p_{in}^j and $i < l < m \leq n$

B.1.1) For all y; if there is a legal path p_{ik}^j for y, propagate y into p_{ik}^j where $k \leq n$.

All events y which are members of local constraint pairs of the current event x are obtained in step A.

Then, a search is conducted in the set of all paths leading from the current transition t_{ij}. This set of paths is denoted by P_i^j. The search determines whether any of the events y already exists in a path p_{in}^j of P_i^j, in step B.1.

If no event y exists in a path p_{in}^j of P_i^j, all events y are to be added. In step B.1.1, they are propagated to all of the legal paths, denoted by p_{ik}^j where $k \leq n$, found in P_i^j.

- *Adding an event that corresponds to end-to-end constraint pair of the current event.*

1.2.1.2.a) Add transitions whose labels correspond to an event in the set of end-to-end constraint pairs of t_{ij} namely $E2EC_{t_{ij}}$ and $E2EC_{t_{ij}^{-1}}$.

The details of this step from the algorithm on page 45 is as follows:

A) Find y in Σ such that $(x, y) \in E2EC_x$ or $(y, x) \in E2EC_{x^{-1}}$ where $x = l(t_{ij})$.

B) if y is a receive event, for all p_{in}^j in P_i^j

B.1) if there is no t_{lm} such that $\exists y = l(t_{lm})$ in p_{in}^j, where $i < l < m \leq n$

B.1.1) find the legal path p_{ik}^j for y, and provided that the local constraints are preserved, propagate y into p_{ik}^j where $k \leq n$.

C) if y is a send event, for p_{ri} (either r=0 provided q_j is not a *"repeated state"* or r=v if $\exists t_{uv} | x = l(t_{uv})$ and $(0 \leq u < v \leq i < j)$, provided q_j is a *"repeated state")*

C.1) if there is no t_{lm} such that $\exists y = l(t_{lm})$ in p_{ri}, where $r \leq l < m \leq i$

C.1.1) provided that the local constraints are preserved, propagate y into the legal path p_{ef} where $v \leq e < f \leq i$.

C.1.2) if no legal path is found or p_{ri} is an empty path (i.e. r=i), provided that the local constraints are preserved, add y *to* the path preceding q_i.

The new event y corresponding to the event pair of the current event is obtained from the *E2EC* in step A. Notice that this step will only yield one event due to Assumption 1 on page 28.

In steps B and B.1, if the event y is a receive event, a search is conducted in all the paths leading from the current transition, P_i^j. This search determines whether the event y already exists in the path following q_j.

In step B.1.1, the event y is propagated to all of the legal paths, denoted by p_{ik}^j, leading from the current transition, t_{ij}, provided the event y does not exist in P_i^j and there is a legal path for the event y such that the local constraint properties of the path are preserved.

If the event y is a send event, a search is conducted to determine the path to be evaluated. This path p_{ri} is either the path from the state whose incoming transition carries the same label x as the current transition t_{ij} (q_j is a "repeated state") or the path from the initial state (q_j is not a "repeated state").

Then, the path p_{ri} is searched for verifying the existence of the event y in step C.1. If this search returns a negative result, the event y is propagated to the legal path p_{ef} preceding the current transition, provided that the local constraint properties of the path are preserved, in step C.1.1.

If p_{ri} is an empty path (r=i) or no legal path is found, the event y is added preceding state q_i, provided that local constraint properties are preserved in step C.1.2. An example of this step can be found on page 124 in Chapter 4.

- *Adding events that correspond to concurrency constraint pairs of the current event.*

1.2.1.3.a) Add transitions which conform to the concurrency constraint pairs of t_{ij} namely $CC_{t_{ij}}$.

The details of this step from the algorithm on page 45 is as follows:

A) Find all y in Σ such that $(x, y) \in CC$ where $x = l(t_{ij})$.

B) if there is no t_{lm} such that $\exists y = l(t_{lm})$ in p_{0i} where $0 \le l < m \le i$

then for all p^j_{in} in P^j_i,

B.1) if there is no t_{lm} such that $\exists y = l(t_{lm})$ in p^j_{in} where $i < l < m \le n$

B.1.1)for each y; if there is a legal path p_{jk} for y,

B.1.1.1) copy p_{jk} to p_{jk}' where q_j is the next state of q_i,

propagate y into p_{jk}', provided that the local constraints are preserved.

B.1.2)otherwise, only add y to q_j (the case where there is no legal path)

If there are any, the events y which correspond to the event pairs of the current event from the CC are found in step A.

Next, a search for the existence of the events y found in step A is conducted on the path p_{0i} leading to the current transition from the initial state in step B.

If the search returns a negative result, another search is conducted on the paths, p^j_{in}, leading from the current transition to verify if the new event already exists in the PSM in any of the paths.

If this second search also yields a negative result (i.e. no event y could be found in any of the paths), the new event y is propagated only to the copies of those paths leading from the current transition p_{jk}' provided that there is a legal path p_{jk} for the new event and the local constraint properties are

preserved, in steps B.1.1 and B.1.1.1.

If there is no legal paths found in step B.1.1 for the new event *y*, the new event *y* is added after the current transition in step B.1.2.

In steps 1.2.1.1.a, 1.2.1.2.a, and 1.2.1.3.a, after adding all the events of the corresponding constraints of the current event, identical paths may have been created due to path propagation effects. Therefore, the amount of new paths to be traversed are reduced by merging the identical paths after processing a transition. This can be done because our model is deterministic. Merging of identical paths is always done from the right of the tree to the left of the tree. Since a pre-order tree traversal algorithm is used, the left of the *PSM* tree is built before the right side of the tree. This is done in steps 1.2.1.1.b, 1.2.1.2.b and 1.2.1.3.b.

In steps 1.2.1.1.c, 1.2.1.2.c and 1.2.1.3.c, the repeated states following another "repeated state" are removed from the PSM because they will never be explored. Again, this can be done because our model is deterministic and a phase has only simple cycles (see Assumption 2 on page 41).

In step 2, the PSM becomes an FSM. This is done in two stages:

- Using the pre-order tree traversal method, a search is conducted to find a state with at least one outgoing transition leading to a "repeated state". Once such a state is found, it is labelled the same as initial state. The paths following this state are not investigated any further. Because these paths are a subset of all the possible event sequences in the previously generated paths starting with the initial events (see Assumption 2 on page 41). This process is repeated until all of the PSM is traversed.
- All paths following the states labelled the same as the initial state are removed.

An example of this step is shown in Figure 28 on page 87 in Chapter 4.

In step 3, all the remaining states can be labelled in the PSM.

3.4.3. Rules for Conservation of Local Constraint Properties

These rules ensure that the local constraints are always satisfied in the PSM. Thus, it prevents the possible corruption of the PSM. All transitions in a path must conform to the following two rules:

Let t_{ij} be a transition on path p_{0n}.

1. If $\exists t_{lm} | SAP(t_{lm}) = SAP(t_{ij})$ and $0 \le l < m \le i < j \le n$, there must be a local constraint pair $(y, x) \in LC$ where $y = l(t_{lm})$ and $x = l(t_{ij})$.
2. If $\exists t_{pr} | SAP(t_{ij}) = SAP(t_{pr})$ and $0 \le i < j \le p < r \le n$, there must be a local constraint pair $(x, z) \in LC$ where $x = l(t_{ij})$ and $z = l(t_{pr})$.

The sequence of events observable from a particular SAP must conform to what is dictated by its set of local constraints.

Often, *projection* is used to retrieve a sequence of events observed at a specific observation point, such as in [Sale 91]. When a path is projected on to a SAP, all other transitions that occur at different SAPs are ignored. The result will be the sequence of events at the SAP which the path was projected onto.

3.4.4. Propagating an Event into a Path

Here, we explain how a new event can be added to a *PSM*. The following rules show how all possible paths are generated with the new event.

Implementation Rules:

Given a legal path p_{ik}, a subtree T_k (if it exists) and a new event e, let

P_i be the set of all possible paths from q_i to q_n,

t_{ij} be the first transition in the path p_{ik},

q_j be the first state that new event e can be added after and

q_k be the last state in the path p_{ik},

For simplicity, it is assumed that $k=n$ when explaining the rules. The case where there are more nodes after q_k is described later in this section. Also, the detailed examples to following rules are given in Chapter 4.

First, the notation used in the rules are clarified:

" $-$ " operator as in $(P_i - p_{ik})$ indicates that path p_{ik} is removed from the set of paths P_i.

"+" operator as in $(P_i + p_{ik})$ indicates that path p_{ik} is added to the set of paths P_i.

"=" operator indicates an assignment.

"Σ"as in $\sum_{m=j}^{l} p_{im}$ refers to a collection paths p_{im} where $j \le m \le l$.

"." concatenates the paths.

In the following figures illustrating the rules, a shaded node depicts a "repeated state".

Rule 1: If $p_k=t_{ij}$ where q_k is not a "repeated state" and $e \neq l(t_{ij})$, i.e. e will not lead to a "repeated state" after it is added to path p_{ik};

$$P_i = (P_i - p_{ik}) + p_{ik} \cdot e$$

In Figure 10 on page 52, it is shown how a new event e is propagated to path p_{ik} using this rule. A new state q' is added to path p_{ik}.

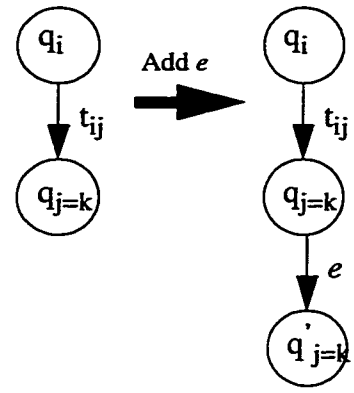


Figure 10. Illustration of rule 1

An example application of this rule is given on page 74 in Chapter 4.

Rule 2: If $p_{ik}=t_{ij}...t_{lk}$ where q_k is not a "repeated state" and $e \neq l(t_{ij})$, i.e. e will not lead to a "repeated state";

$$P_i = (P_i - p_{ik}) + \sum_{m=j}^{l} p_{im} \cdot e \cdot p_{mk} + p_{ik} \cdot e$$

In Figure 11 on page 53, this rule is illustrated. The new event e is propagated to all possible sequence of transitions in the legal path p_{ik}. As a result, a new set of paths is generated. The states q' are the new states added due to this new event.

An example application of this rule can be found on page 76 in Chapter 4.

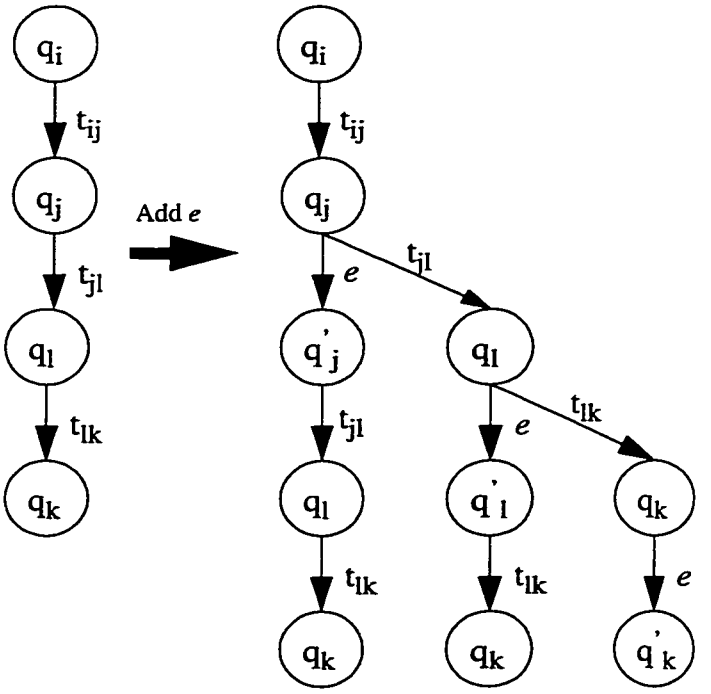


Figure 11. Illustration of rule 2

Rule 3: If $p_{ik}=t_{ij}...t_{lk}$ where q_k is a "repeated state" and $e \neq l(t_{ij})$, i.e. e will not lead to a "repeated state";

$$P_i = (P_i - p_{ik}) + \sum_{m=j}^{l} p_{im} \cdot e \cdot p_{mk}$$

In Figure 12, rule 3 is explained. This rule is very similar to rule 2 except that q_k is a "repeated state". After a "repeated state", there is no need to add a transition since it is cyclic (see definition 1).

An example to this rule is given page 73 in Chapter 4.

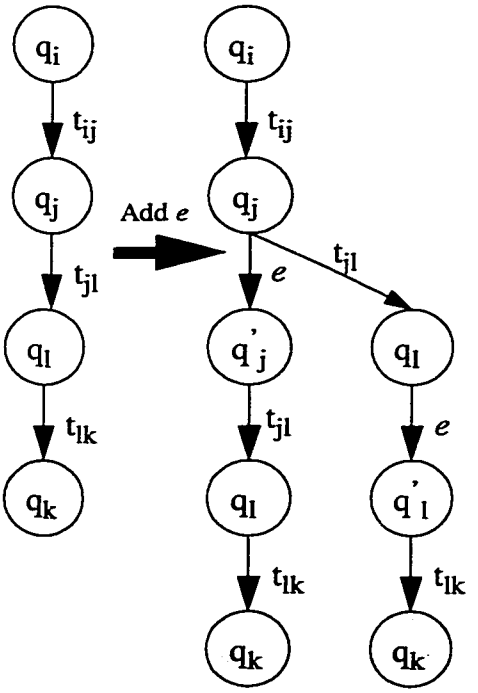


Figure 12. Illustration of rule 3

Rule 4: If $p_{ik}=t_{ij}...t_{lk}$ where q_k is not a "repeated state" and $e= l(t_{ij})$, i.e. e will lead to a "repeated state";

$$P_i = (P_i - p_{ik}) + p_{ik} \cdot e$$

In Figure 13, rule 4 is depicted. In this case, the new event e is added to the end of the path p_{ik} since the new event e will lead to a "repeated state" and paths already generated should be preserved.

An application of this rule can be found on page 73 in Chapter 4.

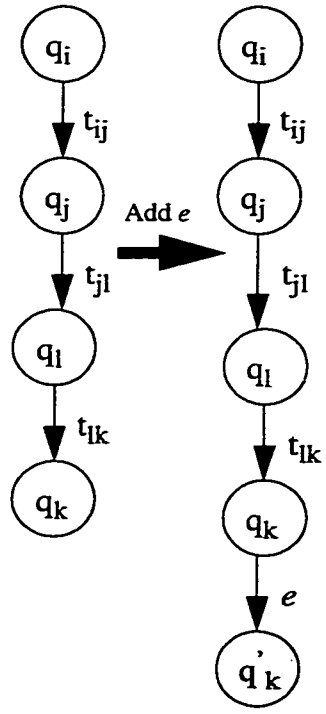


Figure 13. Illustration of rule 4

Rule 5: If $p_{ik}=t_{ij}...t_{lk}$ where q_k is a "repeated state" and $e= l(t_{ij})$, i.e. e will lead to a "repeated state"

$$P_i = (P_i - p_{ik}) + p_{il} \cdot e \cdot p_{lk}$$

In Figure 14, rule 5 is illustrated. This rule mandates what to do if the new event e leads to a "repeated state" and q_k is a "repeated state" as well. The new event e is added before state q_k because the path p_{ik} is a legal path for the new event e.

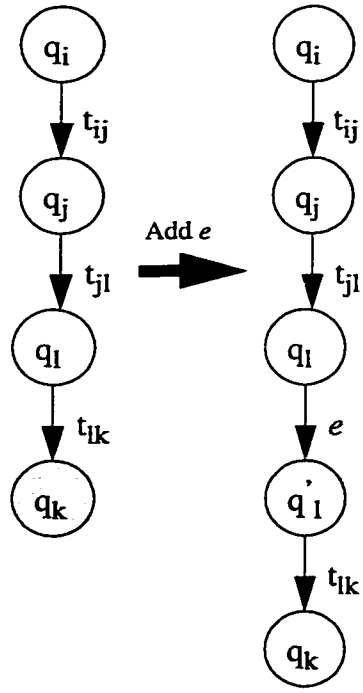


Figure 14. Illustration of rule 5

Given that q_n is the last node in the path which being evaluated and q_k is the last node in the legal path found, propagation also makes sure that:

- prior to propagation if $q_k<q_n$, the remaining path p_{kn} from q_k *to* q_n is copied to all of the newly propagated paths,
- prior to propagation, if there is a subtree after $p_{ik}|(T_k \neq Nil)$, T_k is also copied to all of the newly propagated paths following q_k.

A proof of validity of the propagation rules is not given here. Instead, examples of how the propagation rules work are given in the next chapter as well as in Appendix A.

3.4.5. Optimization Issues and Proposed Improvements for the Algorithm

The current version of the algorithm may explore paths that have a common state one of whose outgoing transitions leads to "repeated state". The algorithm removes such paths after all the paths are discovered since a state is mapped to an initial state (see Assumption 2 on page 41 and Definition 1 on page 42). This is done in step 2 of the "The PSM construction algorithm" on page 44.

A revision of the steps may substantially improve the algorithm. Thus, a change is proposed to add a new step 1.2.1.4 to step 1.2.1; removal of repeated states and labelling of the initial states. Since this will be done after all the constraints of the present transition are evaluated and propagated to the tree, this change will not corrupt the PSM. The algorithm will have to be modified to support this change.

3.5. Step 4: Coupling of Phase Scenario Machines into Global Scenario Machines

In this step, we attempt to couple the *PSM*s that are generated in the previous step. Both other works [Chow 85] and [Choi 86] have addressed the same issues in decomposition terms and developed the same rule for composing a safe network of phases [Chow 85]; i.e. no crossovers or collisions are allowed in the transitions from one phase or partition to another. Recently, [Kaku 94] dealt with a Global Service FSM in synthesizing protocols and added the collision SPs to the Service FSM.

Here, we make use of these three approaches and extend the safety of a network to phases and successfully include the collisions in the resulting PSM. Also, some

exceptional behavior has been identified which can make it difficult to use the method of phase composition proposed by [Chow 85]. Some of these exceptional behaviours are provider-initiated events and deletion of channel contents between the users or reordering them, i.e. not receiving after issuing a T-Disconnect Request at a SAP.

> *Assumption 3:* The communication model used by the synthesis method is explained in Section 3.1.1. on page 19. The channel from $user_i$ to $user_j$ is represented by C_{ij} and the channel from $user_j$ to $user_i$ by C_{ji}. The capacity of the channel between the users is assumed to be one message at a time, i.e. $|C|=1$. Also, the channel contents between the users is assumed to be not modifiable. Thus, we assume totally reliable (lossless) channels.

The above assumption is made to simplify the coupling of the phases.

According to [Chow 85], there exist a network state in each phase in which the network state has information about the states of both protocol entities, the state of the channels between the protocol entities (Figure 3 on page 20), e.g. strings exchanged in the channel. The phase exit state and a phase exit node pair in the network are given. Both entities come to an exit state when they arrive at their final states, i.e. no more outgoing edges. Therefore, all exits from one phase to another (see [Chow 85] for details) can be taken only if the both channels to each protocol entity are empty. This rule prevents collisions from happening at the phase boundaries.

However, service specifications are more flexible and ambiguous than protocol

specifications. We change the exit criteria from one phase to another in order to accommodate more flexibility in the service specification. Hence, we introduce some *collision recovery transitions.*

3.5.1. Obtaining the Interactions at the Phase Boundaries

Given a set of SDs which depict the transitions made from one phase to another, we are interested in those event pairs, one of whose events is an initial event of the phase to be entered. We obtain these pairs by either slicing the SD horizontally or vertically, depending on whether an initial event of the new phase collides with a send event in the present phase. In a collision scenario as seen in Figure 15(c), the SD is horizontally sliced. In a normal scenario, the SDs are sliced vertically. Three examples are given in Figure 15.

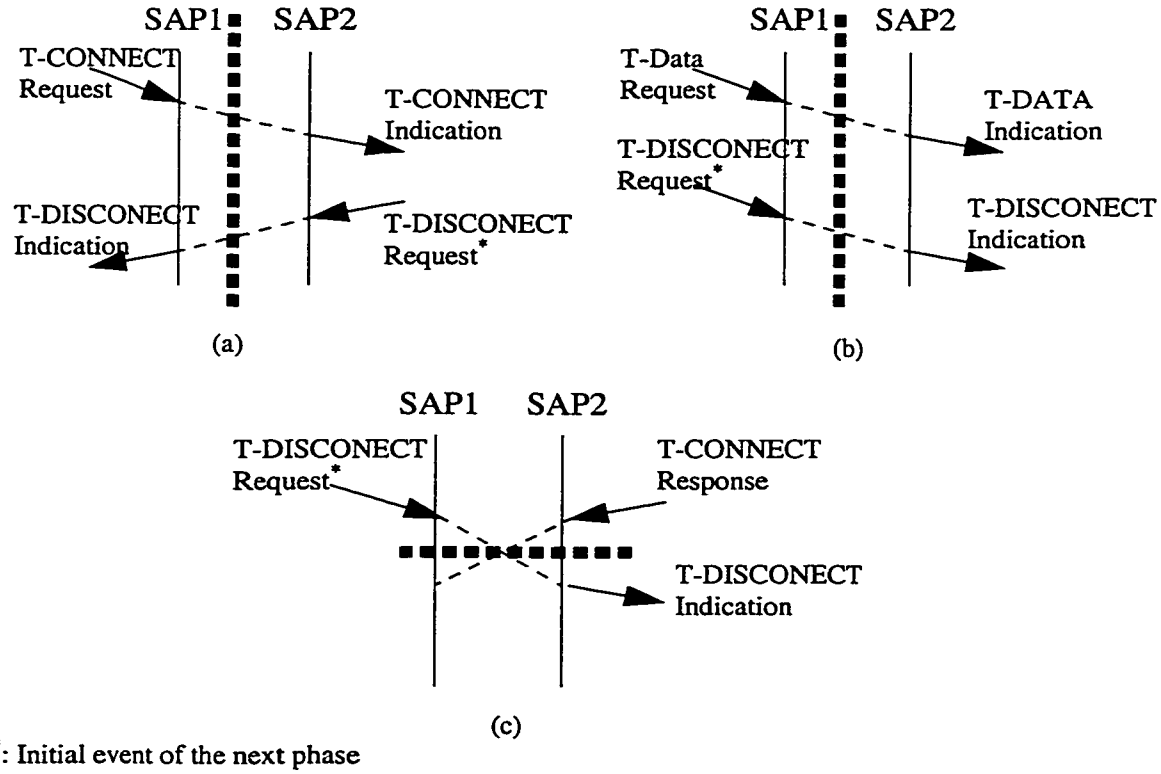


Figure 15. Obtaining the interactions at the phase boundaries

3.5.2. Coupling Criteria and Rules

Similar to the works discussed in Section 3.5, the channel state can be used to couple the phases. The channel capacity is assumed to be $|C_{ij}| = 1$, where i and j refer to the different SAPs and the direction of channel, i.e. from i to j. This is important to simplify the demonstration of how the coupling works. The rules are open to enhancement to accommodate greater channel capacity.

Let C_{ij} and C_{ji} be two channels between SAP_i and SAP_j with directions $SAP_i \rightarrow SAP_j$ and $SAP_j \rightarrow SAP_i$ respectively. Their capacity is assumed to be one message at a time. Given that E is used to indicate the channel is *empty* and M is used to indicate the channel has a *message* in transit, the following are the possible channel states:

- $[C_{ij},C_{ji}]= [E,E]$: In this state, both users are enable to receive events.
- $[C_{ij},C_{ji}]= [M,E]$:In this state, there is a message in transit from SAP_i to SAP_j and the channel from SAP_j to SAP_i is empty. User at SAP_i cannot send any more messages until the present message is received due to the channel capacity. User at SAP_j can receive or send messages.
- $[C_{ij},C_{ji}]= [E,M]$: This state is the same as above with the roles of users reversed.
- $[C_{ij},C_{ji}]= [M,M]$: In this state, both users can only receive messages. No messages can be sent due to the channel capacity.

In Figure 16, the rules for coupling phases is shown. For convenience, send and receive event pairs are shown with the same letter.

When handling phase coupling, it is understood that the exit state of a phase is the

initial state of a new phase [Chow 85].

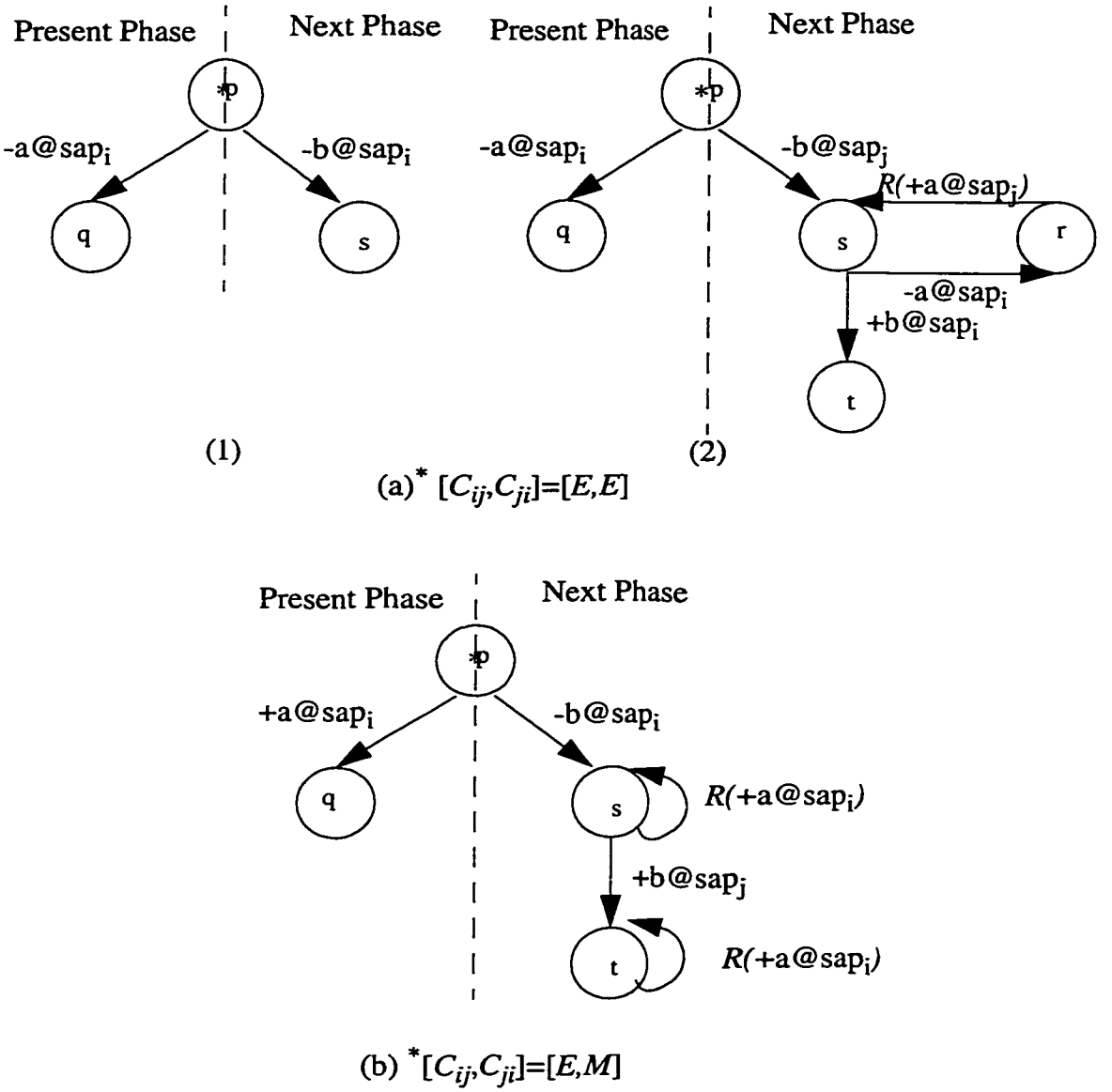


Figure 16. Rules for coupling phases

A new type of service primitive *R()*, (called the *recovery service primitive*), is introduced. Once the service primitive which causes the collision is received, it is encapsulated by this primitive and recovery procedures are executed to bring the service behavior back to normal. The reason for introducing such a concept is that

the protocol specifications have *fault recovery* against the loss of information. A similar approach can be taken for the service specification and the *collision recovery procedures* can be used when specifying the service.

The following are the *coupling rules* (please, refer to Figure 16 for their illustration):

(a) Let q_p be a state on a phase boundary with two outgoing transitions which lead to states q_q in the present phase and q_s in the next phase.

Let us assume that the channel state is empty; $[C_{ij},C_{ji}]=[E,E]$. Since the channels are empty in both directions only send events can be executed. '

Let us also assume that transitions $q_p \rightarrow q_q$ and $q_p \rightarrow q_s$ are both labelled with send events whose service primitives are "a" (belongs to the present phase) and "b" (belongs to the next phase) respectively.

1) If $SAP(l(t_{pq}))=SAP(l(t_{ps}))$, no collisions can happen. Because this is a choice action to be executed by the same entity, coupling can be done with no problem. Please, see Figure 16-a(1).

2) If $SAP(l(t_{pq})) \neq SAP(l(t_{ps}))$, these two events can be issued simultaneously at both SAPs, causing a collision.

In this case, two new transitions are inserted to the next phase to cover the collision scenario. These transitions make sure that a collision is recognized and recovered from, if it occurs.

In addition, a separate state is added to absorb and keep the collision separated from the rest of the service. This is where the collision recovery is done.

As shown in Figure 16-a(2), $-a@SAP_i$ and $-b@SAP_j$ are colliding events. To prevent this collision:

a separate state q_r is created in the next phase,

a transition $q_s \rightarrow q_r$ with label $-a@SAP_i$ is added to recognize the collision and

a transition $q_r \rightarrow q_s$ with label $R(+a@SAP_j)$ is added to indicate the collision is recovered from.

(b) Let q_p be a state on a phase boundary with two outgoing transitions which lead to states q_q in the present phase and q_s in the next phase.

Let us assume that the channel state is $[C_{ij},C_{ji}]=[E,M]$, i.e. a message is in transit from SAP_j to SAP_i. Then, any send event can only be executed from SAP_i to SAP_j.

Also, let $q_p \rightarrow q_q$ be labelled with a receive event at SAP_i "$+a@SAP_i$" (service primitive "*a*" belongs to the present phase) and $q_p \rightarrow q_s$ be labelled with a send event at SAP_j "$-b@SAP_j$" (service primitive "*b*" belongs to the next phase).

In this case, two things could happen;

1) The message in transit is immediately received at SAP_i, e.i. transition $q_p \rightarrow q_q$ is executed. This would not cause any collisions.

2) A send event is executed at SAP_i, i.e. $q_p \rightarrow q_s$ is executed. This would cause a collision with the message, in transit, of the present phase when it is received in the next phase. In this case, a self-looping *collision recovery transition* labelled with *R()* is added all states in sequence from state q_s (including q_s) until a final state or a state whose outgoing transition labelled with a *receive event* at the same SAP (SAP_i) as the *colliding receive event* is found, in order to absorb the collision. In Figure 16-b, this is depicted. Notice that self-looping *collision recovery transitions* labelled with $R(+a@SAP_i)$ are added to states q_s and q_r.

A detailed example of the application of coupling rules is given Section 4.4. on page 89.

3.5.3. Exceptional Cases in Real-Life Services

Although the phase scenario machine construction allows for specifying all of the interactions given by the constraints, allowing the same freedom in coupling phases would require generating a series of all possible scenarios. This would make the service machine quite complicated. There are special cases left out of this step; where two simultaneous send events in the same phase with no reception (e.g. simultaneous T-Connection Release Requests) and handling of spontaneous events (i.e. provider initiated events). These cases are excluded and can be dealt with in future work.

3.6. Techniques for Minimizing the GSM

These techniques reduce a given FSM to a minimal equivalent FSM. A detailed description of these techniques can be found in [Koha 78] or [Linz 90]. These techniques are used to minimize the GSM if it has redundant states.

In the next chapter a detailed case study of the service synthesis method is given. More detailed information can be found in Appendix A.

Chapter 4: Detailed Case Study: Connection-Oriented Transport Service

4.1. Introduction

In this chapter, a detailed case study of the Transport Service, the connection-oriented mode, is given. We illustrate how the synthesis algorithm works by showing snapshots at different stages. More details and a complete example can be found in Appendix A.

First, examples of aids to perform steps 1 and 2 of the method is discussed. Second, the phase scenario machine construction algorithm is explained with examples. Third, an example of our experience in inter-phase coupling is given. Finally, an assessment of the use of this method for the Transport Service is summarized.

The Transport Service is not described here. However, a detailed description of the Transport Service can be found in [ISO 86].

4.2. Preparation of the Data for the Synthesis Method

In Section 3.2 and Section 3.3. on page 36, the type of information given in an informal specification and how to gather data from this specification are discussed. Some examples of the kind of information used to derive the inputs for the synthesis method can be seen in the following pages.

Phases	Service	Primitives	Parameters
TC establishment	TC establishment	T-Connect request	Called address, calling address, quality of service, TS-user data
		T-Connect indication	Called address, calling address, quality of service, TS user-data
		T-Connect response	Quality of service, responding address, TS user-data
		T-Connect confirm	Quality of service, responding address, TS user-data
Data transfer	Normal data transfer	T_Data request	TS user-data
		T-Data indication	TS user-data
TC Release	TC Release	T-Disconnect request	TS user-data
		T-Disconnect indication	Disconnect reason, TS user-data

Table 2: Transport Service

Table 2 is an examples of some information provided in a service specification. As seen in the table, phases, service elements, their associated service primitives and parameters are clearly specified.

In addition to Table 2, a proper sequence of service primitives at a SAP is given in the specification as shown Figure 17 on page 67. The local constraints can easily be gathered from this information.

The TS primitive X / May be followed by the TS primitive Y	T-Connect request	T-Connect confirm	T-Connect indication	T-Connect response	T-Data request	T-Data indication	T-Disconnect request	T-Disconnect indication
T-Connect request								
T-Connect confirm	+							
T-Connect indication								
T-Connect response			+					
T-Data request		+		+	+	+		
T-Data indication		+		+	+	+		
T-Disconnect request	+	+	+	+	+	+		
T-Disconnect indication	+	+	+	+	+	+		

Key:
+: possible
Blank: not possible

Figure 17. Sequences of service primitives allowed [ISO 86]

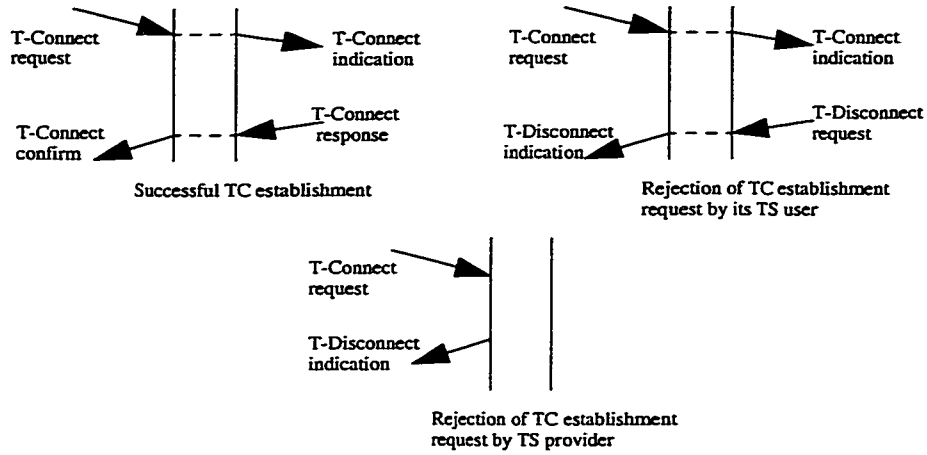
Also, a list of SDs is given to describe the behavior of the users of the service provider or the service provider itself (see Figure 18). An FSM of local service specification and a textual explanation are also provided in the informal Transport Service Specification, [ISO 86].

Although a lot of good and crucial information is given in the informal service specification, our observation is that the global view of the service is not precise.

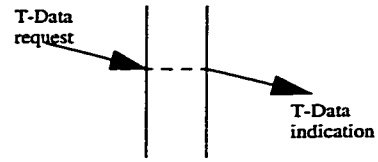
After reviewing the text and SDs (i.e. performing step 1) given in the service specification, a series of SDs are generated in order to complete those interactions which might be missing from the set of SDs given in the service specification.

The list of SDs generated for the Transport Connection Establishment phase is given as an example in Figure 19 on page 70. After reviewing, the text given in [ISO 86] reveals that any user can initiate the Transport Connection Establishment phase. However, this is not shown in the given set of SDs. As a result, additional SDs are generated to reflect this information. A complete set of SDs are generated for the Transport Connection Service is also given in Appendix A.

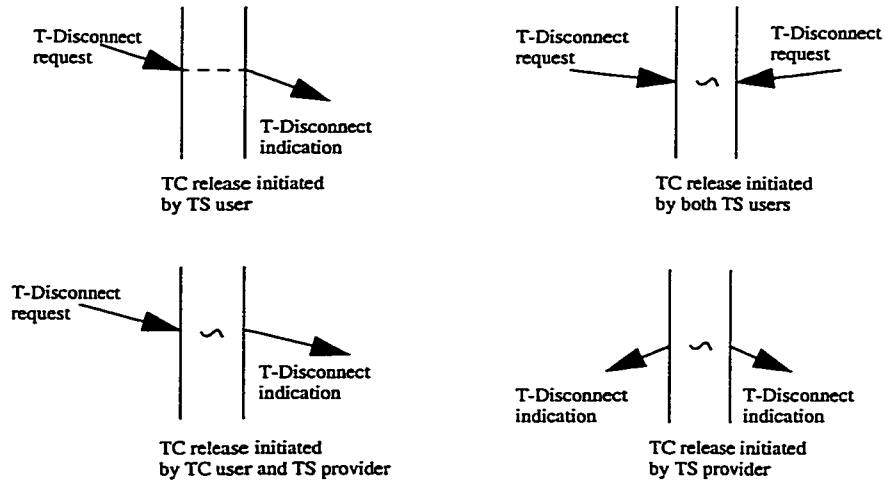
After all input for the service synthesis method is prepared (i.e. performing Step 2), the next step is the Phase Scenario Machine construction. A complete set of inputs can also be found in Appendix A.



(a) Transport Connection Establishment Phase



(b) Data Transfer Phase



(c) Transport Connection Release Phase

Figure 18.Examples of SDs given in [ISO86]

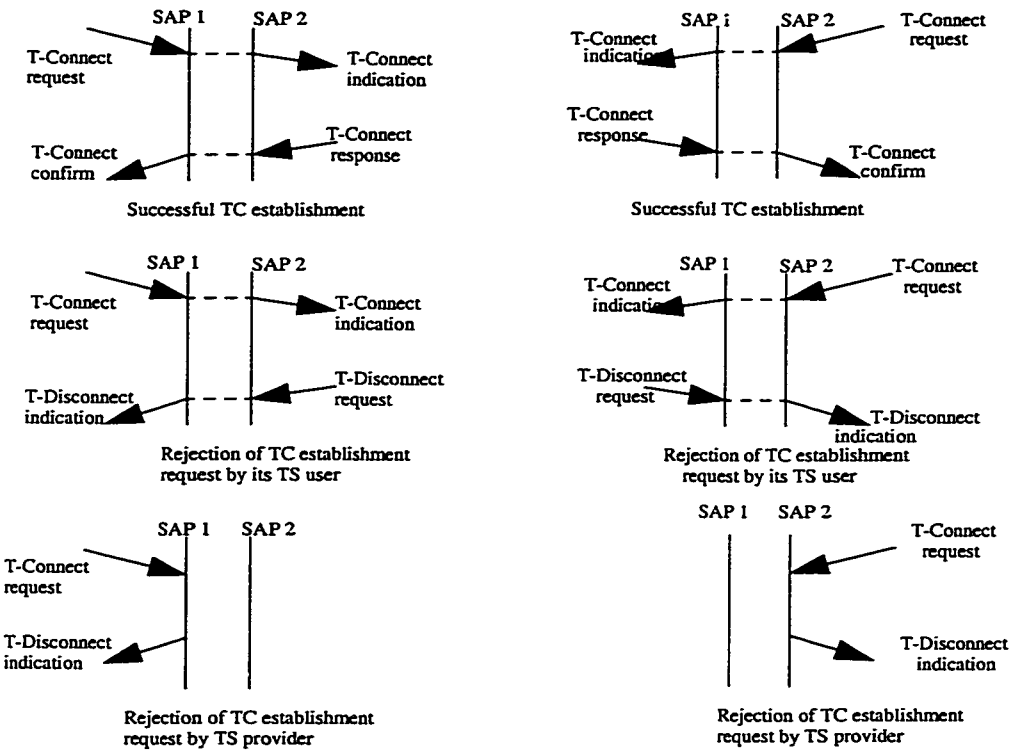


Figure 19. SD for Transports Connection Establishment Phase

4.3. Phase Construction

This step (Step 3) involves building possible scenario paths that can be executed by the service. The constraint pairs are added into a tree one at a time. The end result is a PSM which is a global deterministic CFSM.

Case study for adding local, end-to-end and concurrency constraints to PSM: Here, we give a simple example from the Transport Connection Data Transfer phase. The following data is given in order to construct this phase:

Initial events are -TDataReq@SAP_1 and -TDataReq@SAP_2.

Σ = { -TDataReq@SAP_1, +TDataInd@SAP_2,
-TDataReq@SAP_2, +TDataInd@SAP_1}

LC = { (-TDataReq@SAP_1, -TDataReq@SAP_1),
(-TDataReq@SAP_1, +TDataInd@SAP_1),
(+TDataInd@SAP_1, -TDataReq@SAP_1),
(+TDataInd@SAP_1, +TDataInd@SAP_1),
(-TDataReq@SAP_2, +TDataInd@SAP_2),
(-TDataReq@SAP_2, -TDataReq@SAP_2),
(+TDataInd@SAP_2, -TDataReq@SAP_2),
(+TDataInd@SAP_2, +TDataInd@SAP_2)}

E2EC = { (-TDataReq@SAP_1, +TDataInd@SAP_2),
(-TDataReq@SAP_2, +TDataInd@SAP_1)}

CC = { (-TDataReq@SAP_1, -TDataReq@SAP_2),
(-TDataReq@SAP_2, -TDataReq@SAP_1),
(+TDataInd@SAP_1, +TDataInd@SAP_2),
(+TDataInd@SAP_2, +TDataInd@SAP_1)}

With the above data, we give an example of how to add new transitions conforming to the constraints of the current transition.

In the illustrations of this chapter, a label with a **bold** font denotes the current transition, a label with an underlined font denotes the new transitions that have been added, dark shaded nodes/states denote the repeated states when applicable.

For convenience, all of the states are labelled with numbers to explain the

example.When referring to a specific transition, the state labels such as 1, 2, 3, etc. are used instead of the index of that state such as q_0,q_1,q_2. For example, q_0=1 q_1=2 then $t_{1,2} = 1 \rightarrow 2$ as in Figure 20. This is followed throughout this section when explaining the PSM construction algorithm.

The step numbers given in the following case study refer to the same step numbers as given in "The PSM construction algorithm" on page 44.

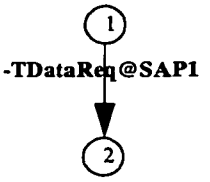


Figure 20.PSM is initialized

Step 1.1:

In this step, the PSM is initialized with an initial event.

Given that initial state is q_0=*1* (state labelled with *1)* and the first initial event is -TDataReq@SAP_1 in the set of initial events;

The PSM is initialized with the first initial event -TDataReq@SAP_1 as seen in Figure 20. State *2* is created and a transition labelled with -TDataReq@SAP_1 is added to the PSM

Step 1.2:

Given that our PSM is in the stage seen in Figure 20, this step calculates the *current transition.*

q_i=q_0= *1*

State *1* is not marked as a *"repeated state"* then the *current transition* is $t_{1,2}$ with $l(t_{1,2})$ = **-TDataReq@SAP_1**.

Step 1.2.1: This step involves steps 1.2.1.1 to 1.2.1.3.

Step 1.2.1.1:

From step 1.2, the *current transition* is $t_{1,2}$ which is labelled with event **-TDataReq@SAP$_1$** in the present PSM seen in Figure 20 on page 72.

Then, $LC_{t_{1,2}}$ is

$LC_{\text{-TDataReq@SAP1}}$ = { (-TDataReq@SAP$_1$, -TDataReq@SAP$_1$),
(-TDataReq@SAP$_1$, +TDataInd@SAP$_1$)}.

a) In this step, after a series of conditions are met (see page 46), the events that correspond to the event pairs in $LC_{\text{-TDataReq@SAP1}}$ are propagated to the PSM.

These events are -TDataReq@SAP$_1$ and +TDataInd@SAP$_1$ which correspond to the event *y* of an event pair *(x,y)* in $LC_{\text{-TDataReq@SAP1}}$.

$P_i^j = P_1^2 = \{p_{1,2}^2\} = \{(t_{1,2})\}$ is the set of all paths to be explored for this step. (See Figure 20 on page 72.)

Thus, there is only one path $p_{1,2}^2$ to be explored:

Step B.1 on page 46 reveals that there is no transition t_{lm} with label -TDataReq@SAP$_1$ or +TDataInd@SAP$_1$ in $p_{1,2}^2$ where $1 < l < m \leq 2$

Therefore, -TDataReq@SAP$_1$ and +TDataInd@SAP$_1$ will be propagated to path $p_{1,2}^2$, provided there is a *legal path* for each event in the path.

- First, -TDataReq@SAP$_1$ is propagated.

Given $p_{1,2}^2$ in the present PSM, legal path $p_{1,2}$ for -TDataReq@SAP$_1$ is found according to Definition 2 on page 43.

When propagating, *propagation rule 4* is used (refer to Rule 4 on page 54) because state *2* is not a *"repeated state"* and the new event -TDataReq@SAP$_1$ is going to lead to a *"repeated state"* after it is

propagated to path $p_{1,2}$ (i.e. event -TDataReq@SAP_1 will be repeated twice in the new path, see Definition 1 on page 42).

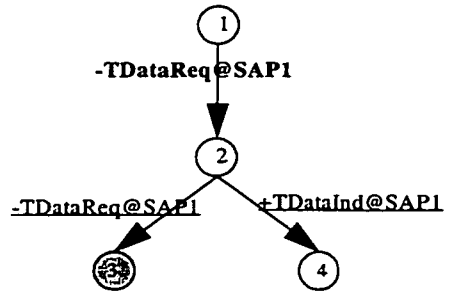
As a result, path $p_{1,3}$ is generated in the new PSM seen in Figure 21 on page 74. A transition with label -TDataReq@SAP_1 is added from state *2* to state *3* which is created and marked as a *"repeated state"* by the propagation rule.

- Second, +TDataInd@SAP_1 is propagated.

Given $p^2_{1,2}$ in the present PSM seen Figure 20 on page 72, legal path $p_{1,2}$ is found for +TDataInd@SAP_1 according to Definition 2 on page 43.

This time, when propagating, *propagation rule 1* is used (refer to Rule 1 on page 51) because state *2* is not a *"repeated state"* and new event +TDataInd@SAP_1 is not going to lead to a *"repeated state"* after it is propagated to path $p_{1,2}$.

As a result, path $p_{1,4}$ is generated in the new PSM seen in Figure 21 on page 74. A transition with label +TDataInd@SAP_1 is added from state *2* to state *4* which is created by the propagation rule.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 21.Adding $LC_{-TDataReq@SAP1}$

b) Since there are no identical paths in the new PSM shown in Figure 21 on page 74, no merging is necessary.

c) There are no two repeated states in sequence in any path. Thus, we do not need to eliminate any "repeated state".

At the end of this step, all the event pairs in set $LC_{\text{-TDataReq@SAP1}}$ exist in new PSM in Figure 21. Next step is to add event pairs in set $E2EC_{\text{-TDataReq@SAP1}}$.

Step 1.2.1.2:

The present PSM is given in Figure 21.

From step 1.2, the *current transition* is $t_{1,2}$ which is labelled with event **-TDataReq@SAP$_1$**.

Then, $E2EC_{t_{1,2}}$ is

$E2EC_{\text{-TDataReq@SAP1}}$ = {(-TDataReq@SAP$_1$, +TDataInd@SAP$_2$)} and

$E2EC_{(\text{-TDataReq@SAP1})^{-1}}$ is an empty set.

There will always be only one *E2EC constraint pair* found for an event because of Assumption 1 on page 28.

a) In this step, after a series of conditions is met (see page 47), event +TDataInd@SAP$_2$ will be propagated to the PSM.

$P_i^j = P_1^2 = \{p_{1,3}^2, p_{1,4}^2\} = \{(t_{1,2} \cdot t_{2,3}), (t_{1,2} \cdot t_{2,4})\}$ is the set of all paths to be explored for this step. (See Figure 21 on page 74.)

In step B on page 47, it is found that +TDataInd@SAP$_2$ is a *receive event.* Therefore, it should exist in the paths following the *current transition* $t_{1,2}$.

First, path $p_{1,3}^2$ is explored:

Step B.1 on page 47 reveals that there is no transition t_{lm} in $p_{1,3}^2$ with label +TDataInd@SAP$_2$ where $1 < l < m \leq 3$.

Thus, +TDataInd@SAP$_2$ will be propagated to path $p_{1,3}^2$.

Given $p_{1,3}^2$, legal path $p_{1,3}$ for +TDataInd@SAP$_2$ is found according to

Definition 2 on page 43.

When propagating, *propagation rule 3* is used (refer to Rule 3 on page 53) because state *3* is a *"repeated state"* and the new event +TDataInd@SAP_2 is not going to lead to a *"repeated state"* after it is propagated to path $p_{1,3}$.

As a result, path $p_{1,4}$ is generated in the new PSM seen in Figure 22 on page 77. The new transition added in the path $p_{1,4}$ is shown with an underlined label.

Next, path $p^2_{1,4}$ in the present PSM seen in Figure 21 is explored:

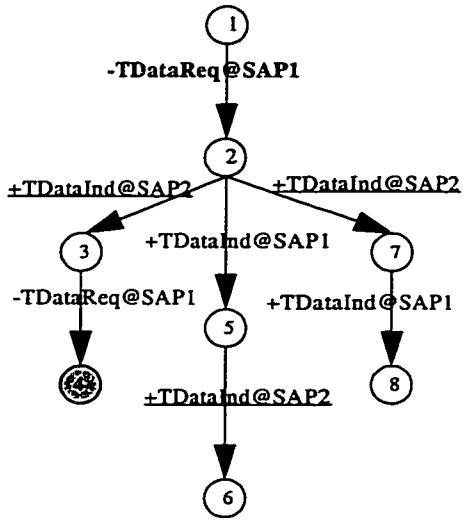
Step B.1 on page 47 reveals that there is no transition t_{lm} in $p^2_{1,4}$ with label +TDataInd@SAP_2 where $1 < l < m \leq 4$.

As a result, +TDataInd@SAP_2 will be propagated to path $p^2_{1,4}$.

Given $p^2_{1,4}$, legal path $p_{1,4}$ is found for +TDataInd@SAP_2 according to Definition 2 on page 43.

When propagating, *propagation rule 2* is used (refer to Rule 2 on page 52) because state *4* is not a *"repeated state"* and the new event +TDataInd@SAP_2 is not going to lead to a *"repeated state"* after it is propagated to path $p_{1,4}$.

As a result, paths $p_{1,6}$ and $p_{1,8}$ are generated in the new PSM shown in Figure 22 on page 77. The transitions with the underlined labels are the new transitions added in paths $p_{1,6}$ and $p_{1,8}$.



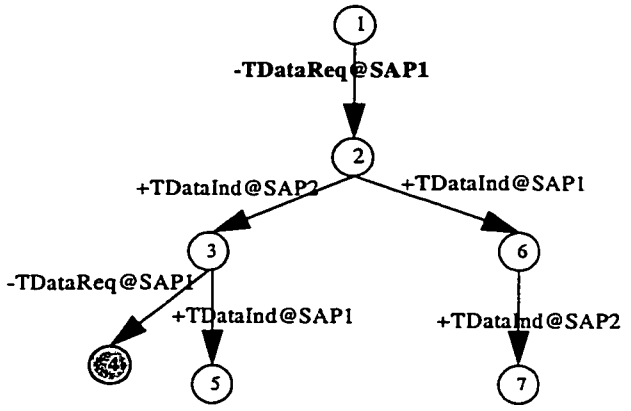
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 22.Adding E2EC$_{-TDataReq@SAP1}$

b) Notice that identical paths are generated due to propagation.

As explained in Section 3.1.3. on page 22, our PSM model is deterministic. Therefore, no two outgoing transitions from a state with the same label can exist.

As a result, identical paths are merged at this step, and the new PSM is shown in Figure 23.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 23. After merging identical transitions in the PSM

c) There are no two "repeated state" in sequence in any path. Therefore, we do not need to eliminate any "repeated state".

At the end of this step, the new PSM satisfies $E2EC_{\text{-TDataReq@SAP1}}$. Next step is to add event pairs in set $CC_{\text{-TDataReq@SAP1}}$.

Step 1.2.1.3:

The present PSM is given in Figure 23.

From step 1.2, the *current transition* is $t_{1,2}$ which is labelled with event **-TDataReq@SAP$_1$**.

Then, $CC_{t_{1,2}}$ is

$CC_{\text{-TDataReq@SAP1}} = \{(\text{-TDataReq@SAP}_1, \text{-TDataReq@SAP}_2))\}$.

a) In this step, after a series of conditions is met (see page 48), event -TDataReq@SAP$_2$ will be propagated to the PSM.

$$P_i^j = P_1^2 = \{p_{1,4}^2, p_{1,5}^2, p_{1,7}^2\} = \{(t_{1,2} \cdot t_{2,3} \cdot t_{3,4}), (t_{1,2} \cdot t_{2,3} \cdot t_{3,5}), (t_{1,2} \cdot t_{2,6} \cdot t_{6,7})\}$$

is the set of all paths to be explored for this step (see Figure 23 on page 77).

In Step B on page 48, a search is done to verify whether -TDataReq@SAP$_2$ exist in the path preceding the current transition. Although it does not seem logical to perform this search because we are dealing with the initial transition, it is explained to show how the algorithm works. Since concurrent events in our PSM is expressed as interleaved events, -TDataReq@SAP$_2$ could theoretically have been in the preceding path of state *1*. The search yields that there is no transition with label -TDataReq@SAP$_2$ in path $p_{1,1}$ where $1 < l < m \leq 1$. Then, the next step is to explore the set of paths P_1^2.

First, path $p_{1,4}^2$ is explored:

Step B.1 on page 48 reveals that there is no transition t_{lm} in $p^2_{1,4}$ with label -TDataReq@SAP_2 such that $1 < l < m \leq 4$.

Therefore, -TDataReq@SAP_2 is to be propagated to the copy of path $p^2_{1,4}$.

Given $p^2_{1,4}$, the legal path $p_{1,4}$ for -TDataReq@SAP_2 is found according to Definition 2 on page 43.

When propagating, *propagation rule 3* is used (refer to Rule 3 on page 53) because state *4* is a *"repeated state"* and new event -TDataReq@SAP_2 is not going to lead to a *"repeated state"* after it is propagated to the copy of path $p_{1,4}$.

In addition to the propagation rules, each transition of a path generated by propagation is checked against local constraints as explained in Section 3.4.3. Rules for Conservation of Local Constraint Properties on page 50.

As a result, new paths $p_{1,10}$ and $p_{1,13}$ are generated in the new PSM seen in Figure 24 on page 81. The transitions with underlined labels are the new transitions added in paths $p_{1,10}$ and $p_{1,13}$. Note that path $p_{1,4}$ is still unchanged in the new PSM. Propagation is done only to the copy of $p_{1,4}$.

Next, path $p^2_{1,5}$ is explored:

Step B.1 on page 48 reveals that there is no transition t_{lm} in $p^2_{1,5}$ with label -TDataReq@SAP_2 such that $1 < l < m \leq 5$.

Therefore, -TDataReq@SAP_2 is to be propagated to the copy of path $p^2_{1,5}$.

Given $p^2_{1,5}$, legal path $p_{1,3}$ is found for -TDataReq@SAP_2 according to Definition 2 on page 43.

When propagating, *propagation rule 2* is used (refer to Rule 2 on page 52) because state *3* of the *legal path* is not a *"repeated state"* and the new event -TDataReq@SAP_2 is not going to lead to a *"repeated state"* after it is propagated to the copy of path $p_{1,3}$.

In addition to the propagation rules, each transition of a path generated by propagation is checked against local constraints as explained in Section 3.4.3. Rules for Conservation of Local Constraint Properties on page 50.

As part of the propagation as mentioned on page 50, since the *legal path* is a subset of the actual path being explored, a copy of the remaining path $p_{3,5}$ is also appended to the newly propagated paths.

As a result, new paths $p_{1,16}$ and $p_{1,19}$ are generated in the new PSM seen in Figure 24 on page 81. The transitions with underlined labels are the new transitions added in paths $p_{1,16}$ and $p_{1,19}$.

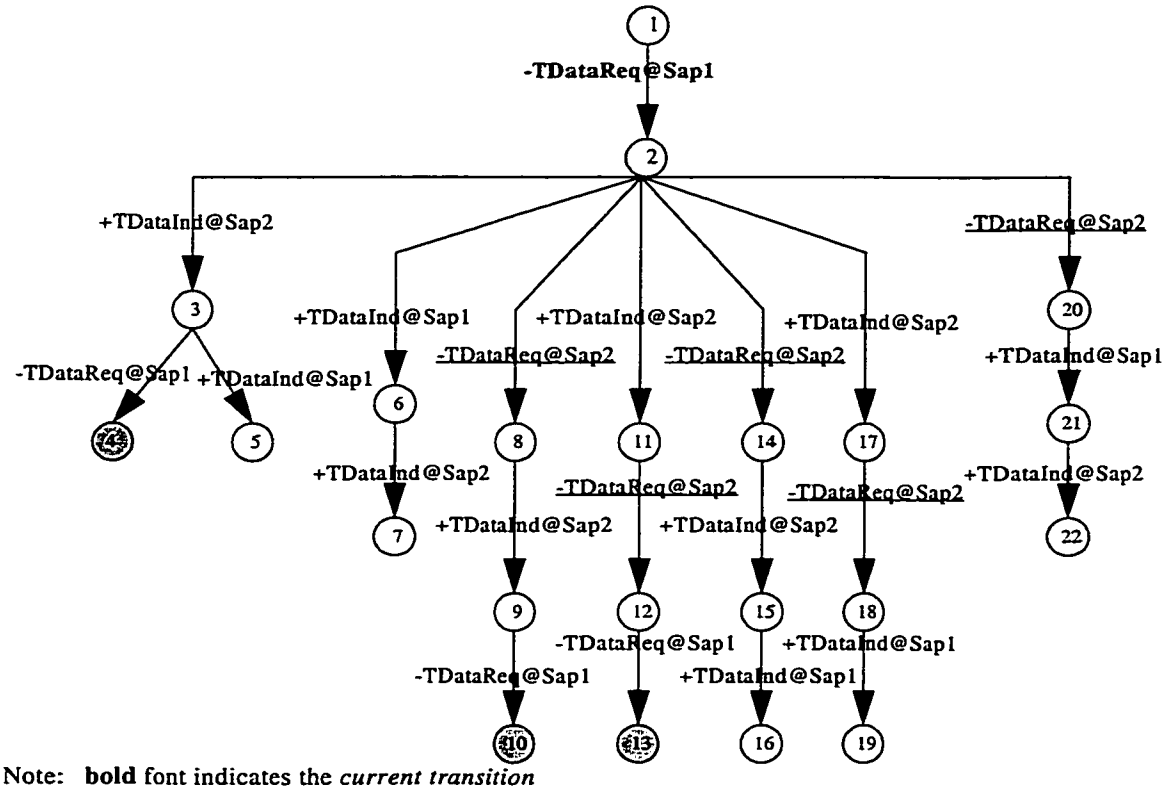
Last, path $p^2_{1,7}$ is explored:

Step B.1 on page 48 reveals that there is no transition t_{lm} in $p^2_{1,7}$ with label -TDataReq@SAP_2 such that $1 < l < m \leq 7$.

Therefore, -TDataReq@SAP_2 is to be propagated to the copy of path $p^2_{1,7}$.

Given $p^2_{1,7}$, the legal path $p_{1,2}$ is found for -TDataReq@SAP_2 according to Definition 2 on page 43.

When propagating, *propagation rule 2* is used (refer to Rule 2 on page 52) because state *2* of the *legal path* is not a *"repeated state"* and the new event -TDataReq@SAP_2 is not going to lead to a *"repeated state"* after it is propagated to the copy of path $p_{1,2}$.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 24.After adding new transitions correspond to CC constraint pairs of -TDataReq@Sap1

In addition to the propagation rules, each transition of a path generated by propagation is checked against local constraints as explained in Section 3.4.3. Rules for Conservation of Local Constraint Properties on page 50.

As part of propagation as mentioned on page 50, since the *legal path* is a subset of the actual path being explored, a copy of the remaining path $p_{2,7}$ is also appended to the newly propagated path.

As a result, the new path $p_{1,22}$ is generated in the new PSM seen in Figure 24. The new transition added is indicated with an underlined label in path $p_{1,22}$.

Figure 24 shows the new PSM after propagation. Notice how large it can get

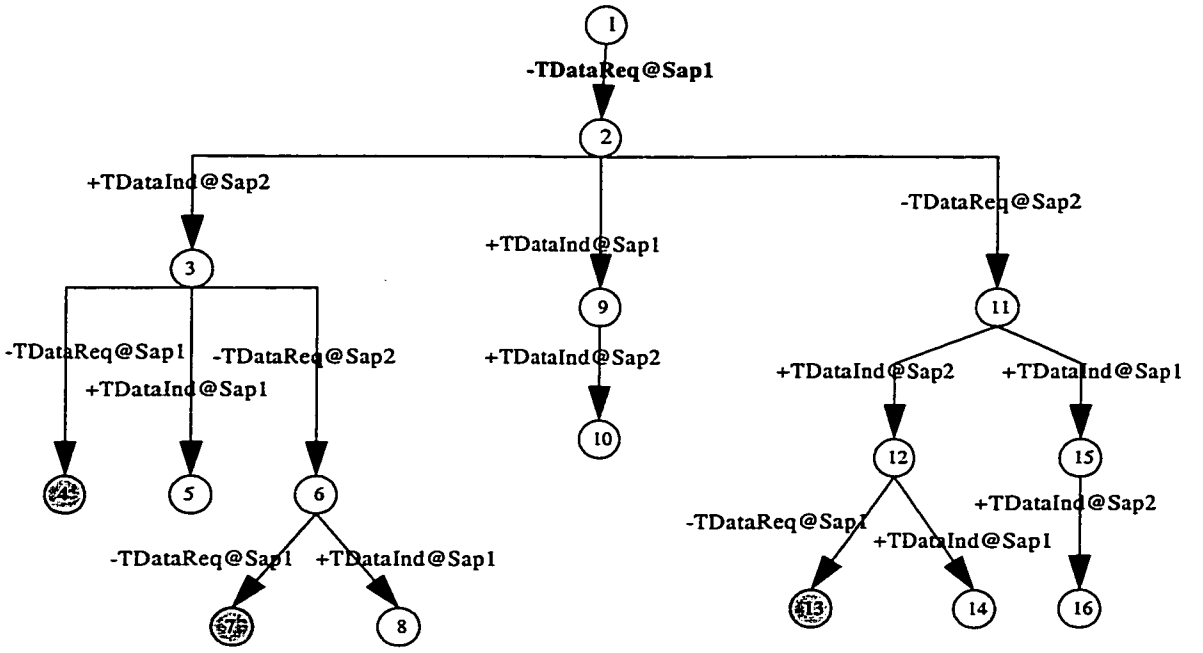
suddenly.

At the end of this step, the new PSM satisfies $CC_{\text{-TDataReq@SAP1}}$. All the constraints of event -TDataReq@SAP_1 are satisfied.

b) Notice that identical paths are generated due to propagation.

As explained in Section 3.1.3. on page 22, our PSM model is deterministic. Therefore, no two outgoing transitions from a state with the same label can exist.

As a result, identical paths are merged at this step, and the new PSM is seen in Figure 25.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 25.After merging the identical paths in the PSM

c) There are no two repeated states in sequence in any path. Therefore, we do not need to eliminate any "repeated state".

The next step is to go back to step 1.2 and assign a new transition to the current transition, given the present PSM in Figure 25 on page 82. Then, all of sub-steps of step 1.2 are executed once more. This sequence is repeated until all the transitions are traversed in the PSM. The tree traversal algorithm used is the pre-order tree traversal algorithm as described in [Aho 74].

Notice that we did not give an example in this chapter for Step B.1.2. of *1.2.3.a* described on page 43. In this particular step, where no legal paths are found, only a transition labelled with the new event would be added separately following the current transition and a new state whose incoming transition is this newly added transition would be created, i.e. the new event is not propagated to any existing path but a new path is started with this new event. An example of this step is given in Figure 47 on page 142 in Appendix A when adding *CC constraint pair* (-TDiscReq@SAP_1, -TDiscReq@SAP_2).

So far, we have illustrated the uses of propagation rules 1, 2, 3 and 4. However, propagation rule 5 requires more repeated states to be generated because of the following condition: a new transition is going to lead to a *"repeated state"* after it is propagated to a path and the path already has a *"repeated state"* as its leaf node. In order to illustrate *propagation rule 5*, the following example is given.

Illustration of propagation rule 5:

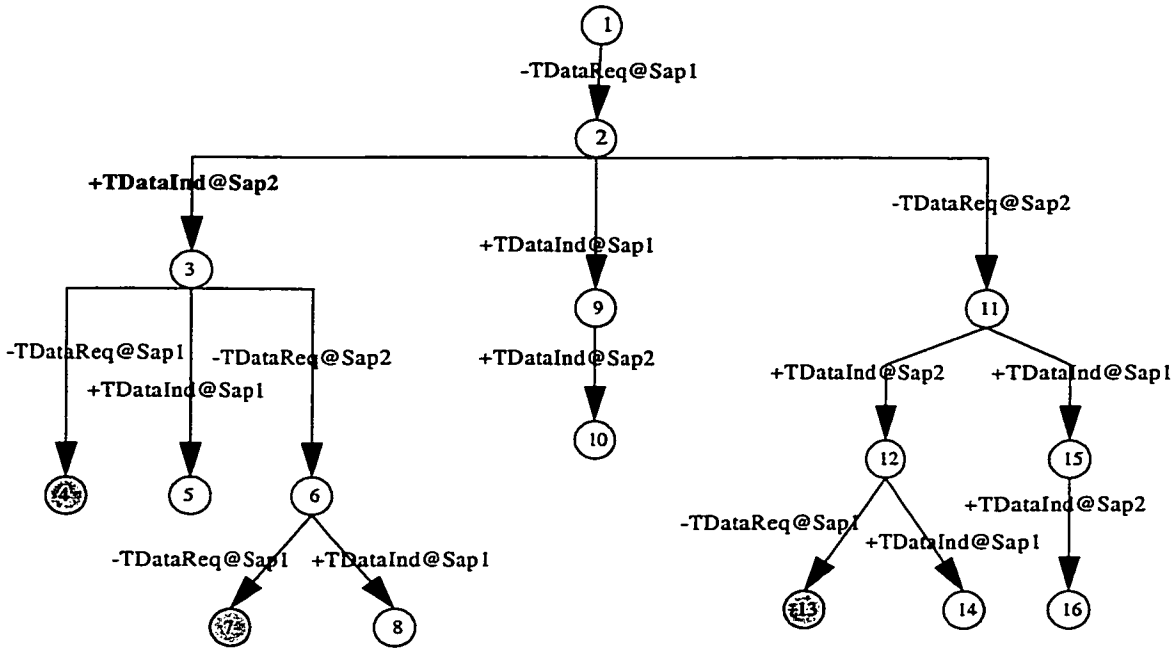
This example covers the next execution of steps from 1.2 to 1.2.1.1, given the present PSM in Figure 25 on page 82.

In step 1.2, the current transition is assigned as transition $t_{2,3}$ labelled with event **+TDataInd@SAP$_2$** according to pre-order tree traversal algorithm (see Figure 26 on page 84).

Next step 1.2.1.1 is executed:

$LC_{t_{1,2}}$ is

$LC_{+\text{TDataInd@SAP2}}$ = { (+TDataInd@SAP$_2$, -TDataReq@SAP$_2$),
(+TDataInd@SAP$_2$, +TDataInd@SAP$_2$)}.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 26.the new current transition assigned in the PSM

a) In this step, after a series of conditions is met (see page 46), the events that correspond to the event pairs in $LC_{+\text{TDataInd@SAP2}}$ are propagated to the PSM. These events are -TDataReq@SAP$_2$ and +TDataInd@SAP$_2$ which are the y in an event pair (x,y) in $LC_{+\text{TDataInd@SAP2}}$.

$$P_i^j = P_2^3 = \{p_{2,4}^3, p_{2,5}^3, p_{2,7}^3, p_{2,8}^3\} =$$
$$\{(t_{2,3} \cdot t_{3,4}), (t_{2,3} \cdot t_{3,5}), (t_{2,3} \cdot t_{3,6} \cdot t_{6,7}), (t_{2,3} \cdot t_{3,6} \cdot t_{6,8})\}$$

is the set of all paths to be explored for this step.

First, path $p_{2,4}^3$ to be explored:

Step B.1 on page 46 reveals that there is no transition t_{lm} with a label -TDataReq@SAP_2 or +TDataInd@SAP_2 in $p_{2,4}^3$ where $2 < l < m \leq 4$.

Therefore, -TDataReq@SAP_1 and +TDataInd@SAP_1 will be propagated to path $p_{2,4}^3$, provided there is a *legal path* for each event in the path.

- First, -TDataReq@SAP_2 is propagated using *propagation rule 3*. The details of the propagation are not discussed since the *rule 3* is explained with an example prior to this one. As a result, path $p_{2,5}$ is generated in the new PSM seen in Figure 27 on page 86.

- Next, +TDataInd@SAP_2 is propagated.

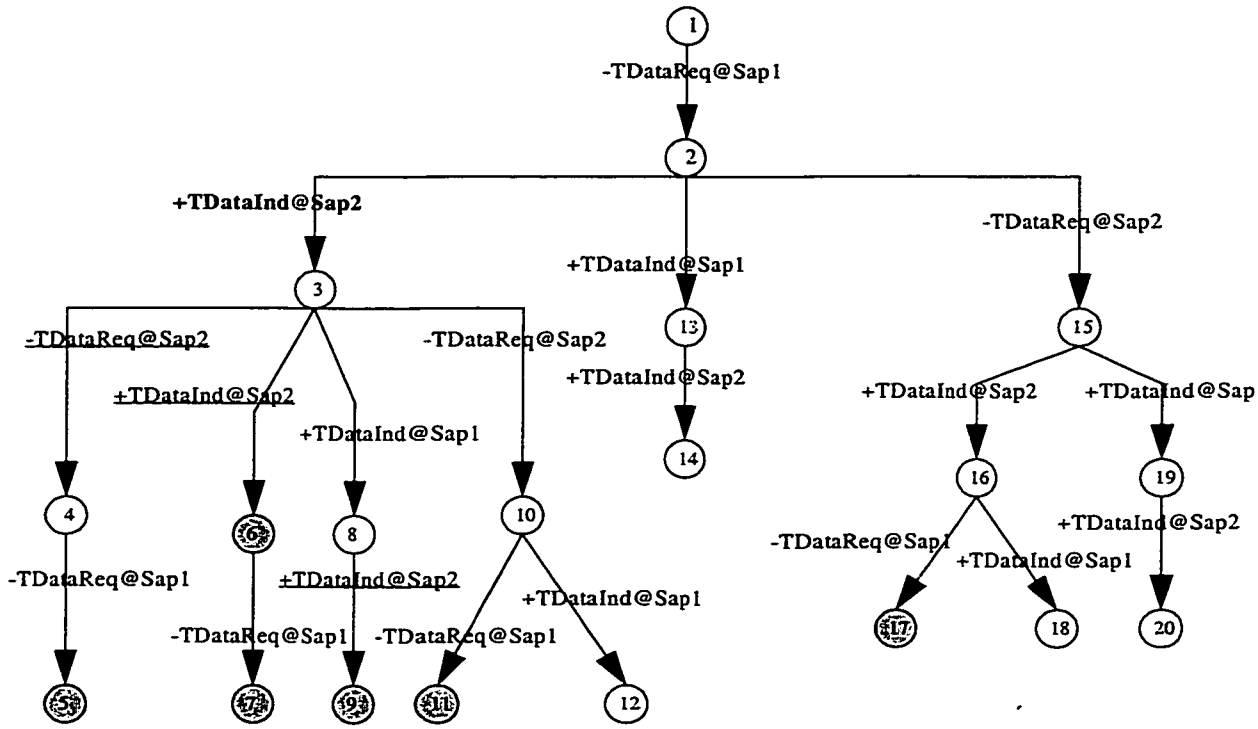
  Given $p_{2,4}^3$ in the present PSM, legal path $p_{2,4}$ for +TDataInd@SAP_2 is found according to Definition 2 on page 43.

  When propagating, *propagation rule 5* is used (refer to Rule 5 on page 55) because state *4* is a *"repeated state"* and the new event +TDataInd@SAP_2 is going to lead to a *"repeated state"* after it is propagated to path $p_{2,4}$ (i.e. event +TDataInd@SAP_2 will be repeated twice in the new path, see Definition 1 on page 42).

As a result, path $p_{2,7}$ is generated in the new PSM seen in Figure 27 on page 86. A transition with label $\underline{+TDataInd@SAP_2}$ is added from state *3* to state *6* which is created and marked as a *"repeated state"* by the propagation rule. Notice that state *6* and state *7* are repeated states.

Then, the remaining paths are explored one at a time. The details are not given since each of the *propagation rules* is explained with an example so far.

b) After all the new paths are generated, the identical transitions are merged. The resulting PSM is shown in Figure 27.



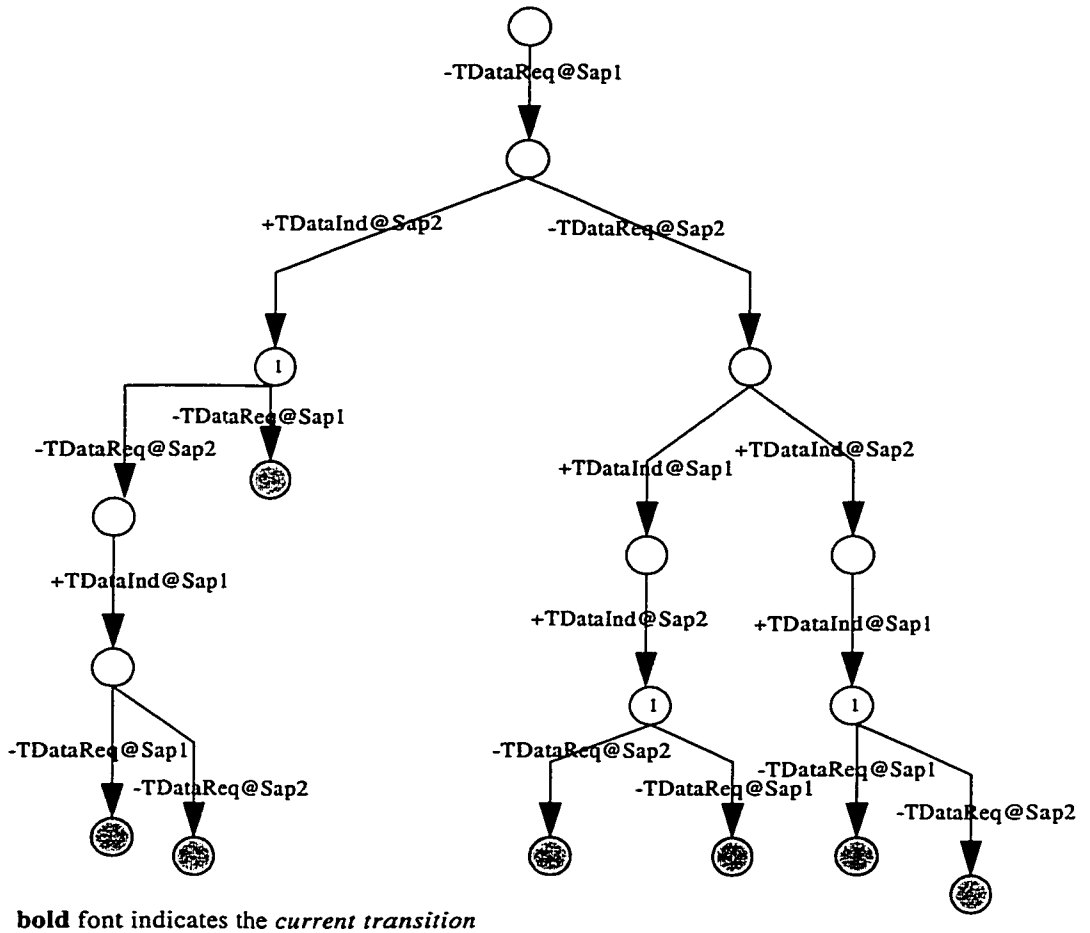
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 27. Illustration of propagation rule 5

c) At this step, if there are two consecutive repeated states in a path, the last "repeated state" is removed from the path. Note that path $p_{2,7}$ has two repeated

states, *6* and *7*. Therefore, state *7* and its incoming transition are to be removed from the path.

In Figure 28, the resulting PSM is generated for initial event -TDataReq@SAP_1 after all sub-steps of Step 1 are executed for all the transitions in the PSM.



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Figure 28. The Transport Connection Data Transfer Phase after it is constructed

The PSM seen in Figure 28 is one half of the final PSM to be generated from Step 1 of the PSM construction algorithm. The other half, generated from -TDataReq@SAP_2, is symmetric to the one in Figure 28. The symmetry comes

from the fact that both users can initiate a Data Transfer phase. Since the PSM builds each phase separately, the complexity of such cases is minimized. For simplicity, Steps 2 and 3 of the Algorithm (see page 45) are demonstrated on this PSM shown in Figure 28 on page 87.

Step 2: As shown in Figure 28, the states looping back to the initial state are identified in this step. This is done by looking at all of the outgoing transitions of each state in the PSM, starting with the initial state. If anyone of the outgoing transitions of a state leads to "repeated state", then this state is labelled the same as the initial state. In this case, they are labelled with *1*. Then all the following paths of these states are removed. The new PSM is shown in Figure 29; it looks more like a global deterministic CFSM.

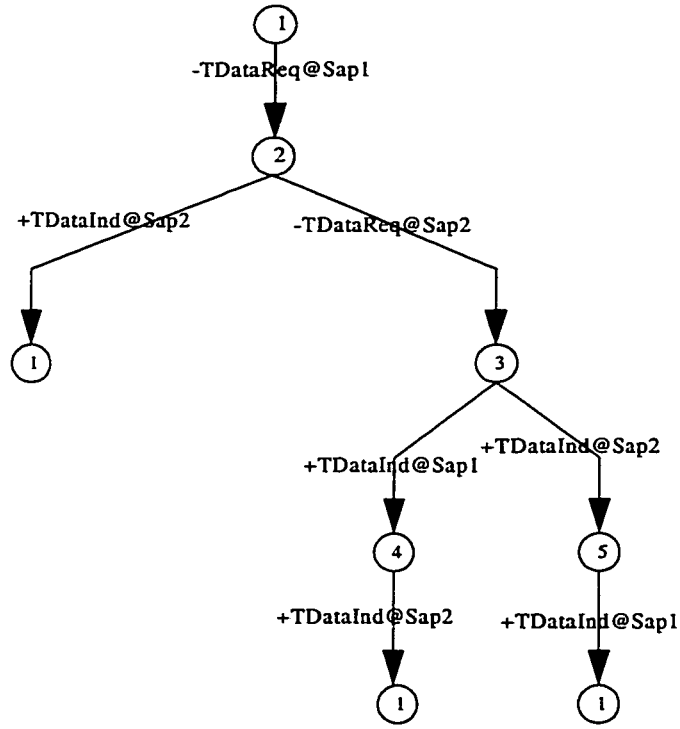


Figure 29. PSM for Transport Connection Data Transfer Phase after its states are labelled

Step 3: The states are labelled as in Figure 29. Those states that are equivalent to

the initial state are labelled the same as the initial state in this step.

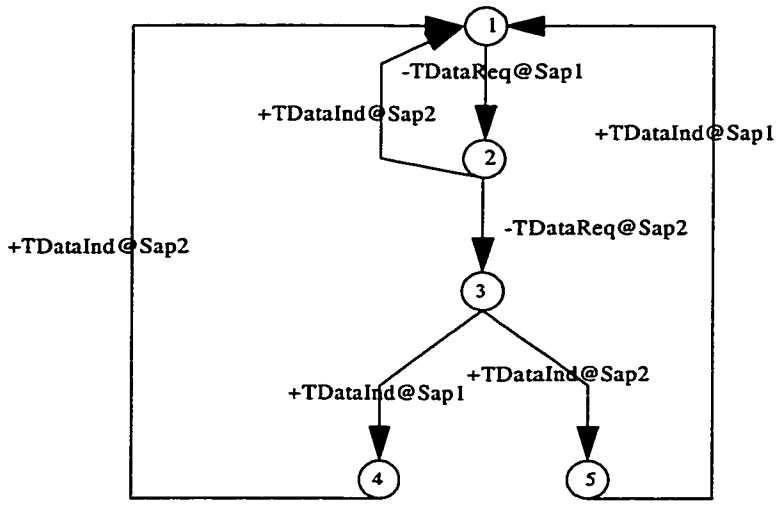


Figure 30. The final PSM for the Transport Connection Data Transfer Phase

In Figure 30, the final PSM for the Transport Connection Data Transfer phase is shown after all the states with the label of the initial state is connected to the initial state.

4.4. Coupling of Phase Scenario Machines

Here, we give an example of how the two phases can be coupled at their phase boundaries. For this part of the case study, we use the Transport Connection Data Transfer Phase and the Transport Connection Release Phase.

First, a very important assumption has to be made to illustrate the coupling. We assume the users cannot issue a T-Disconnect Request concurrently at the same time since their execution means the deletion of the channel contents [ISO 86].

The method can be extended to support such exceptions. However, this kind of event is not supported presently. To illustrate the method, we assume that provider-initiated events are not allowed either.

The following is a set of event pairs between phases as discussed above:

{ (-TDataReq@SAP_1, -TDiscReq@SAP_2),

(-TDataReq@SAP_1, -TDiscReq@SAP_1),

(-TDataReq@SAP_2, -TDiscReq@SAP_1),

(-TDataReq@SAP_2, -TDiscReq@SAP_2),

(+TDataInd@SAP_1, -TDiscReq@SAP_1),

(+TDataInd@SAP_2, -TDiscReq@SAP_2)}

The channel states of the Transport Connection Data Transfer Phase Scenario Machine (see Figure 30 on page 89) are given in Table 3.

State	C_{12}	C_{21}
1	Empty	Empty
2	Msg	Empty
3	Msg	Msg
4	Msg	Empty
5	Empty	Msg

Table 3:

For coupling of the Transport Connection Data Transfer Phase to the Transport Connection Release Phase, an analysis of each state of the PSM shown in Figure 30 on page 89 is given below. In order to understand the coupling examples, it

can be referred to Appendix A for the specification of the Transport Connection Release PSM:

State 1: The label of the outgoing transition of state *1* is -TDataReq@SAP_1. From Table 3, both channels are *empty* when the PSM is in state *1*. This means either user can issue a send event from each end of the channel. According to the set of event pairs given on page 90, -TDataReq@SAP_1 is included in two event pairs. The resulting coupling is seen in Figure 31. In this case, two different types coupling is done:

(a) is a collision scenario where two users independently execute events -TDataReq@SAP_1 and -TDiscReq@SAP_2 in different phases. The collision can happen and the service can recover if it can recognize them. In the case (a), a way of handling the collisions, i.e. including them in the service specification is given. This is an example of the coupling rule (a)-2 shown in Figure 16 on page 61.

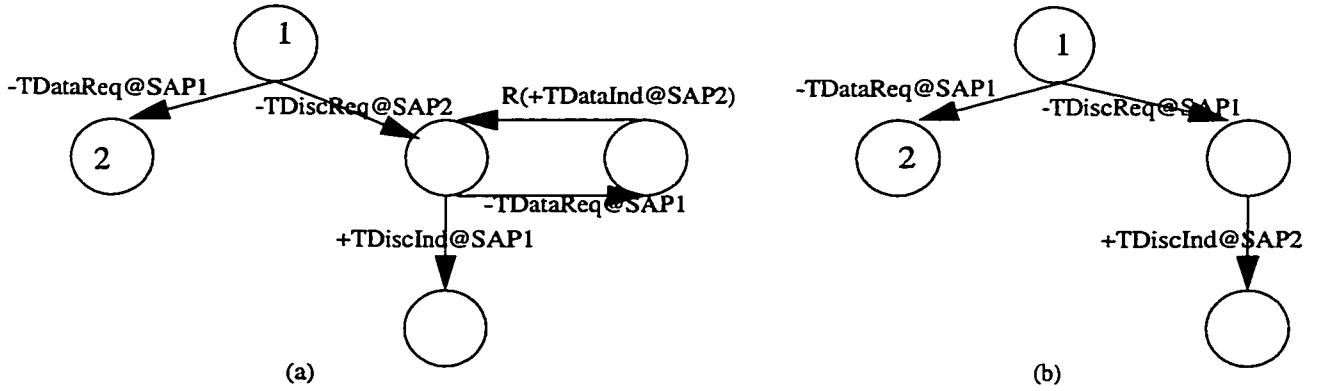


Figure 31. Coupling at State 1

(b) is choice case. The same user has the choice of executing events

-TDataReq@SAP_1 or -TDiscReq@SAP_1 in either phase.

State 2: State 2 has two outgoing transitions with labels; +TDataInd@SAP_2 ($t_{2,1}$) and -TDataReq@SAP_2 ($t_{2,3}$). From Table 3, the channel state is [*M,E*]. This means a send event can only be issued at SAP_2 for transition to the next phase. Hence, coupling can only be done for event pairs (+TDataInd@SAP_2, -TDiscReq@SAP_2) and (-TDataReq@SAP_2, -TDiscReq@SAP_2). The result is shown in Figure 32.

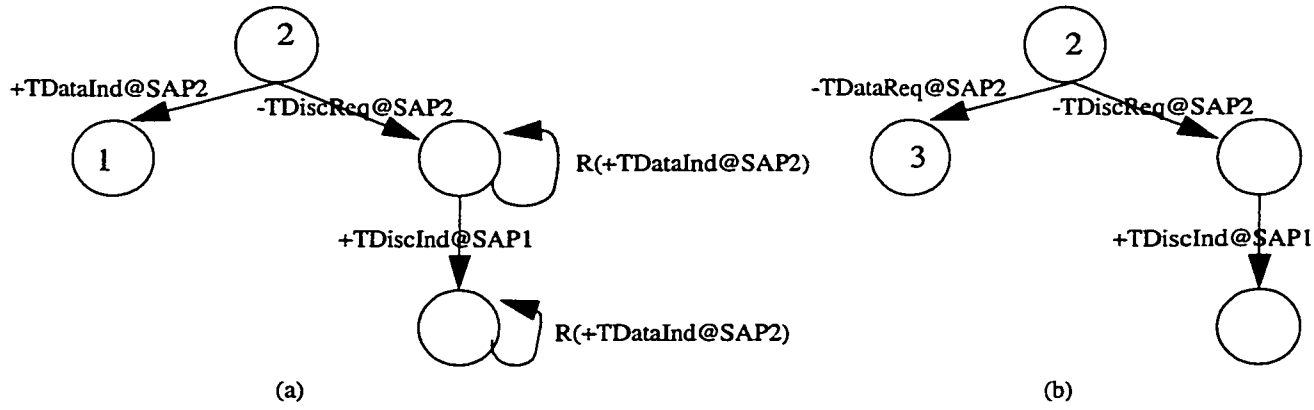


Figure 32. Coupling at State 2

(a) is the case where there is a pending receive event in one channel and the other channel is empty. Again, the collisions are recognized by the next phase, should they happen. This is an example of *coupling rule b-2* described on page 63.

(b) is a choice case and no collision can happen.

State 3: From Table 3, the channel state is [*M,M*]. Therefore, no users can send

any messages for transition to the next phase.

State 4: State *4* has only one outgoing transition labelled +TDataInd@SAP_2. From Table 3, the channel state is [*M,E*]. The coupling of this state is similar to coupling of state *2*, see case (a) in Figure 32 on page 92. The same rules will apply when coupling at this state as the ones used when coupling at state *2* with event +TDataInd@SAP_2. Figure 33 depicts the coupling at state *4*.

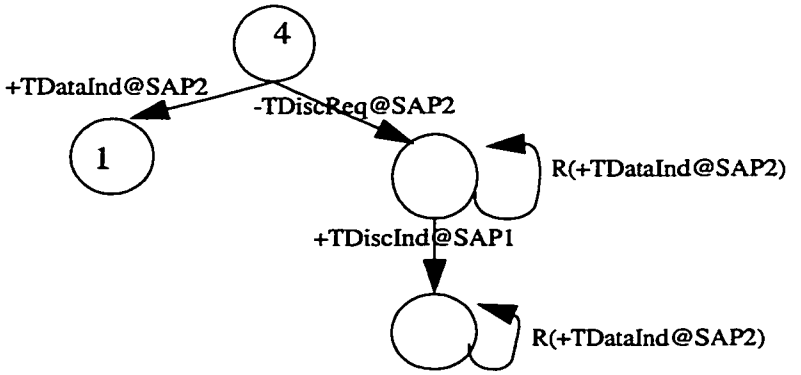


Figure 33. Coupling at State 4

State 5: State *5* has only one outgoing transition labelled +TDataInd@SAP_1. From Table 3, the channel state is [*E,M*]. Once more, this case is very similar to state *4*. Therefore, it will not be explained in detail. The result of the coupling is seen in Figure 34.

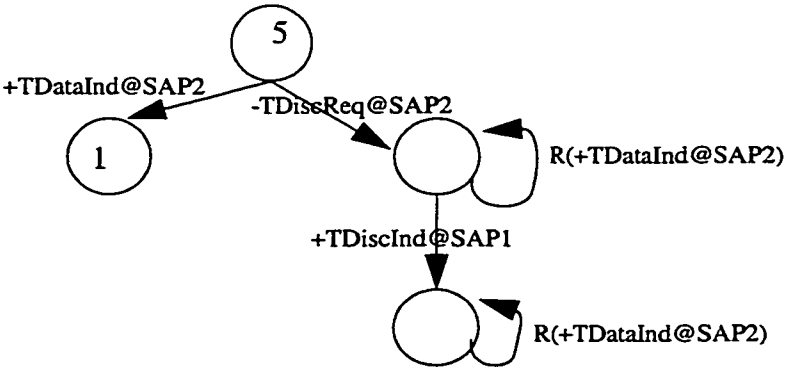


Figure 34. Coupling at State 5

In this part of the case study, an example for each *coupling rule* is given. Next, the overall assessment of the service synthesis method is discussed.

4.5. A Brief Assessment of the synthesis method

The PSM construction algorithm is very good since it can produce all possible sequence of events within a phase. It can be costly if the identical paths are constantly produced, recognized and eliminated. Also the elimination of the consecutive repeated states add to the work involved.

Phases that do not have any repeating paths can be constructed without much effort. The number of paths grows linearly with the number of transitions identified in a legal path. Each time an event is propagated to the PSM, the merging of identical paths is done. The effect of this is minimal.

The simultaneous send events are handled well in phase scenario machine construction. However, when channel states are considered and channels are assumed to be *reliable* and *undestroyable*, the freedom given in the stages of phase construction cannot be given here, to keep the illustration of the method simple.

The coupling of phases has limitations since all the exceptions cannot be handled freely. A similar approach to [Chow 85] is taken when coupling. The coupling method explained here is a good example for handling collisions. A contribution is made by introducing collision recovery transitions.

Chapter 5: Conclusions

5.1. Contributions of the Thesis

The main contribution of this thesis is a new method for synthesis of services from a combination of natural language descriptions and constraints. A number of detailed contributions are included to make up this method.

There are two main steps in our new Service Synthesis Method; the construction of the Phase Scenario Machines and their coupling. Detailed contributions are indicated below via italics.

a) Phase Scenario Machine Construction. This thesis describes a method for generating a Phase Scenario Machine (PSM) which represents all the scenarios defined by a set of constraints within a phase. The method is *automatable* and the resulting specification represents the *global behavior* of the service over a set of Service Access Points (SAPs). A listing of the program that implements this method can be found in Appendix B.

There are *three* types of constraints used: *local, end-to-end* and *concurrency constraints*. The first two types of constraints have been used before, such as in

[Boch 80] and [Faci 91]. The use of *concurrency constraints* is a contribution of this thesis. These constraints identify events occurring at different SAPs which could occur in either order. These events are called relatively concurrent [Yu 92], and should be taken into account during design. A restriction is imposed on the end-to-end constraints; there is a one-to-one relation between a message sent a SAP and a message received as a consequence at a remote SAP. While this is a restriction, it is not a serious one since most services such as Transport Service have this property. Another limitation is that underlying communication medium between users is made of two FIFO channels (one in each direction). Thus, the PSM construction algorithm does not cover cases where there is prioritized messaging or there are more than two channels between two service users. However, the communication model used in this thesis is also used by many other studies such as [Chow 85], [Sale 91] and [Choi 86].

Also, we show how to partition a Sequence Diagram (SD) to extract these constraints systematically.

Collisions originating from the users executing the same phase of the service are expressed in an interleaved representation. The *exceptional cases* such as collision of disconnection primitives and service provider initiated events (i.e. spontaneous events*)* are represented in the *phase specification*.

The use of *constraints* instead of partial scenarios allows identification of situations that sometimes cannot be represented with just one scenario. It also helps for identifying unexpected situations which were not represented in a

scenario. This can be done because the set of all possible sequences of behavior is generated.

While in a tree-form, the resulting *Phase Scenario Machine* can be used for *validation of phases*.

Thus, this step produces additional useful information about the service in a structured representation.

b) Coupling of Phase Scenario Machines. The *phase coupling method* introduced here has limitations. Again, the communication medium between the service users is assumed to be reliable FIFO channels with message capacity one where the channel contents cannot be changed. This type of assumption is common [Mill 90]. It eases the coupling of phases. However, some exceptional cases that were expressed in the PSM have to be eliminated, i.e. removal of channel contents (for example, handling of simultaneous disconnection requests) and spontaneous events.

A contribution is made to the work of [Chow 85], by defining and capturing explicit *collision recovery transitions*. When executed, these transitions assure the service returns to the normal mode of operation. A set of rules which is given to identify where to insert these transitions.

Collision recovery transitions can also be used on already defined protocol or service specifications in which the composite or global behavior is represented. This can be done because the rules only rely on the channel state between the

users.

Thus, this step extends previous work in this direction.

Another contribution is the *case study* in detail. This provides some real experience with the use of the approach. In Appendix A, all the details of the case study is given, including the step by step generation of the PSMs and coupling of PSMs. In Appendix B, an early version prototype implementation of the PSM construction algorithm is given.

5.2. Comparison to Related Work

In chapter 2, other service synthesis techniques [Hsia 94], [Dsou 95] and [Sale 96] are discussed.

In [Hsia 94], only the local view of services specification is generated. Here, we generate the global service specification. In [Hsia 94], the method assumes only one type of cycles may exist in the service specification, i.e. all looping transitions must loop back to the initial state. Therefore, the method can only describe the limited behavior. Our method relaxes this assumption to such cycles existing in a phase rather than in the complete service specification.

All service synthesis methods referred here are incremental and they result in a single service machine. We approach the synthesis problem from a modular view. This allows the service to achieve modularity.

Our work attempts to represent concurrent events. Concurrent events are not dealt

with in [Hsia 94]. [Dsou 95] assumes there are only independent parallel events which do not affect each other. In [Sale 96], concurrency has been discussed but the algorithm does not include concurrent events.

In [Kaku 94], the collisions are identified and included in the service FSM. Our method introduces a *collision recovery transition* and a set of rules to insert these transitions to the service specification. To the best of my knowledge, no other service synthesis method has included the *collision recovery transitions* at the time of writing.

5.3. Future Work

The algorithm for the coupling of phases needs to be enhanced. To remove the limitations of the coupling of phases, an approach similar to the PSM construction can be adapted instead of using only the *channel states* introduced as the coupling criteria. By this, we mean that a set of constraints which can be called the *coupling constraints* can be used for coupling.

During this work, it became clear that there is a need for a Constraint Specification Language. The behavior can be described by a set of constraints and the Service Specification can be built using it.

5.4. Summary

We have described a methodology for building service from informal specifications and SDs. The method is semi-automated since the designer has to extract the constraint sets from the informal service specification. At this time,

the channel states are also analyzed by the designer, however, this could easily be automated.

Appendix A. Details and Case Study: Connection-Oriented Transport Service

A.1. Analysis of the Informal Service Specification

In this case study, we will use the transport service. A detailed description of the Transport Service can be found in [ISO 86]. The following information has been given in [ISO 86]:

- The phases of the transport service, their service primitives used within each phase and the parameters associated with each service primitive in a table form (see Table 4).
- The partial scenarios in the form of Sequence Diagrams (SD) (see Figure 36).

Phases	Service	Primitives	Parameters
TC establishment	TC establishment	T-Connect request	Called address, calling address, quality of service, TS-user data
		T-Connect indication	Called address, calling address, quality of service, TS user-data
		T-Connect response	Quality of service, responding address, TS user-data
		T-Connect confirm	Quality of service, responding address, TS user-data
Data transfer	Normal data transfer	T-Data request	TS user-data
		T-Data indication	TS user-data
TC Release	TC Release	T-Disconnect request	TS user-data
		T-Disconnect indication	Disconnect reason, TS user-data

Table 4:

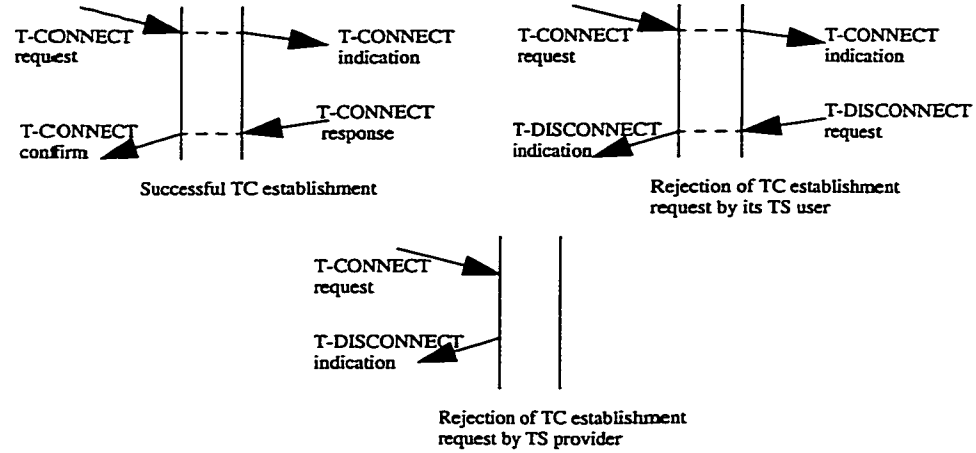
- The local constraints for a single Transport connection at one end have been in a table form (see Figure 35).

- The local view of Transport Service in a Finite State Machine (FSM) form to describe the possible allowed sequences of service primitives at a Transport connection endpoint.

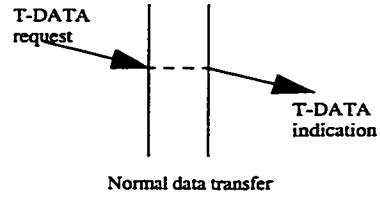
The TS primitive X / May be followed by the TS primitive Y	T-Connect request	T-Connect confirm	T-Connect indication	T-Connect response	T-Data request	T-Data indication	T-Disconnect request	T-Disconnect indication
T-Connect request								
T-Connect confirm	+							
T-Connect indication								
T-Connect response			+					
T-Data request		+		+	+	+		
T-Data indication		+		+	+	+		
T-Disconnect request	+	+	+	+	+	+		
T-Disconnect indication	+	+	+	+	+	+		

Key:
+: possible
Blank: not possible

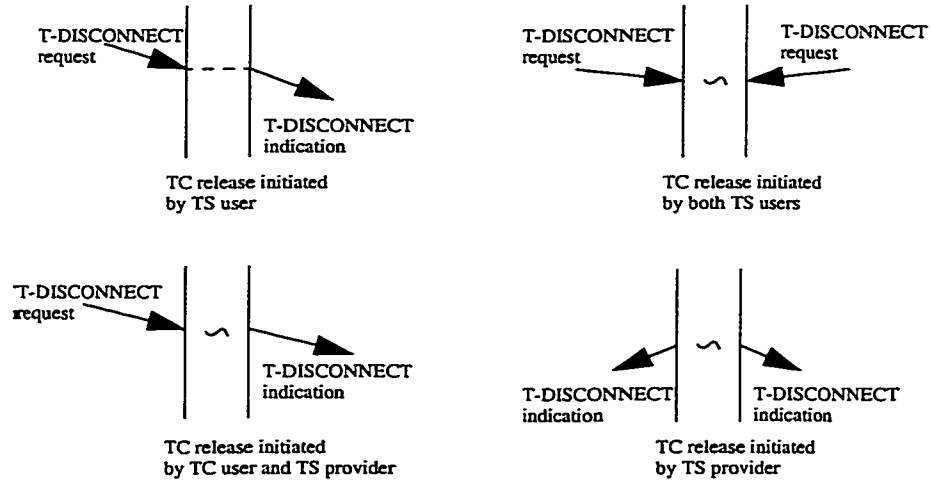
Figure 35. Local sequences of service primitives allowed [ISO 86]



(a) Transport Connection Establishment Phase



(b) Data Transfer Phase



(c) Transport Connection Release Phase

Figure 36. End-to-end constraints of Transport Service as it is given in [ISO 86]

A.2. Some Additional SDs

After analyzing the specification, the following are the examples of derived information not reflected in the Sequence Diagrams (SDs):

- Either user can issue a T-Connect Request.
- Either user can transfer data.
- A user can issue a T-Disconnect Request at any time.

As a result, the new set of SDs for each phase can be shown as in Figure 37, Figure 38 and Figure 39.

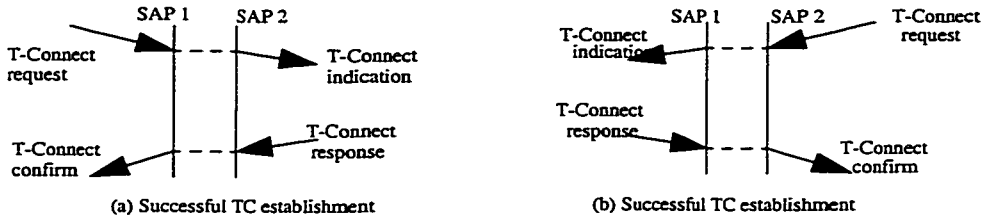


Figure 37.Complete set of SDs for the Transport Connection Establishment phase

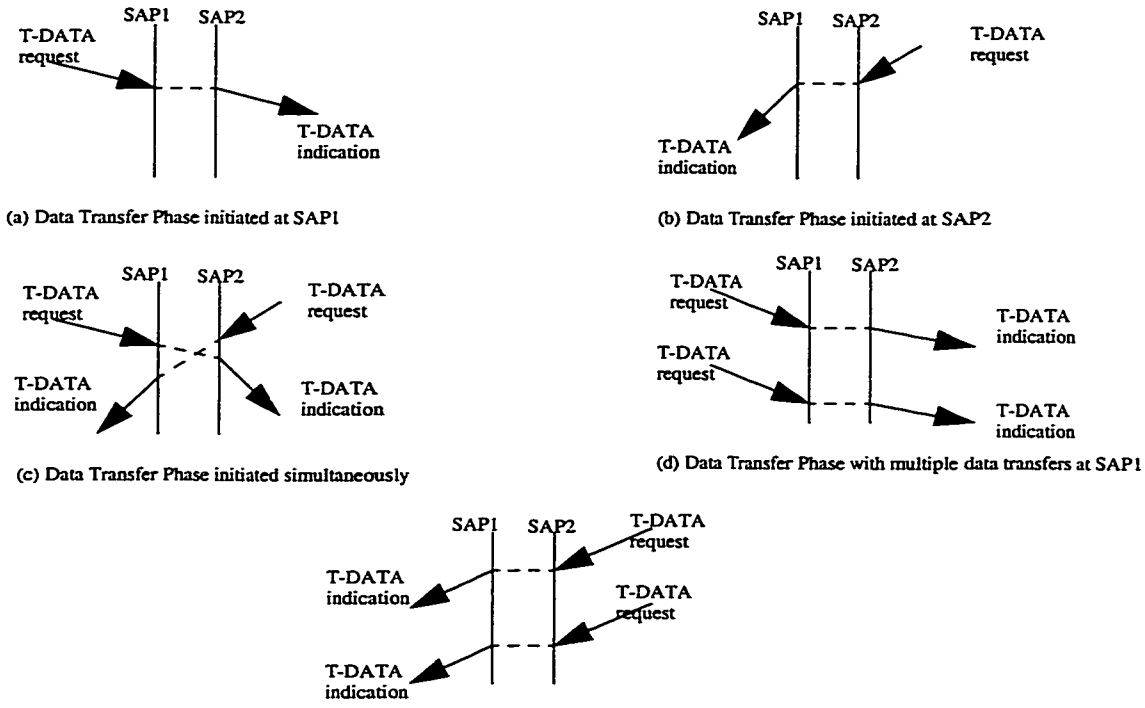


Figure 38. Complete set of SDs for the Transport Connection Data Transfer phase

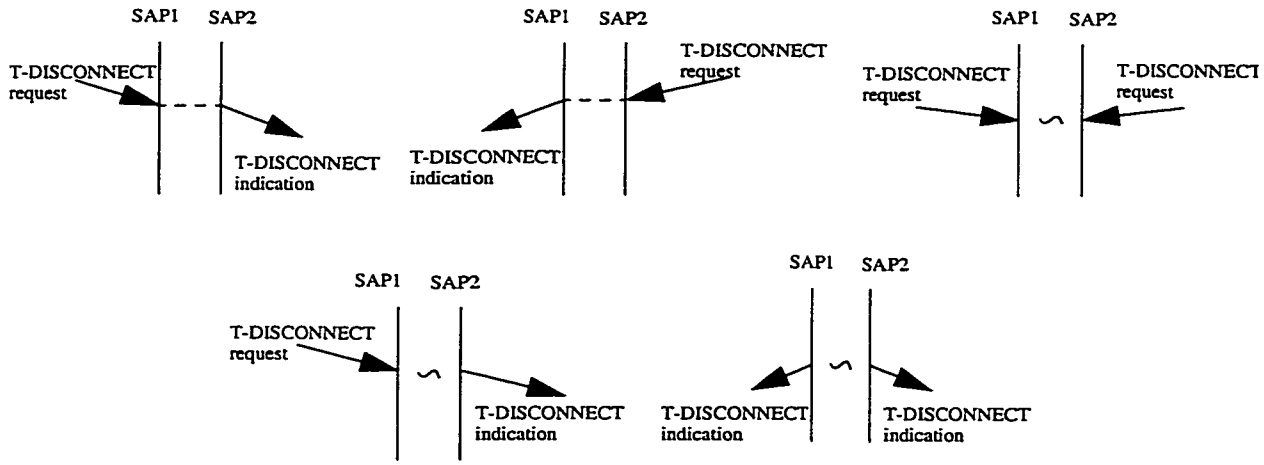


Figure 39. Complete set of SDs for the Transport for Transport Connection Release Phase

A.3. Phase Scenario Machine Construction

After finding the complete set of SDs and identifying the inputs, i.e. initial events, set of events, set of local, end-to-end and concurrency constraints for each phase, the Phase Scenario Machines (PSM) can be constructed.

This section attempts to illustrate how the PSM is built by giving snapshots of the PSM during the execution of the PSM construction algorithm. Note that the steps which do not produce any new paths are not shown explicitly. In the following figures, a label with a **bold** font denotes the current transition, a label with an underlined font denotes the new transitions that have been added, dark shaded nodes/states denote the repeated states when applicable. For the symbols and definitions used in this section, please refer to Chapter 3. For detailed walk through of the steps of the algorithm, please refer to Chapter 4.

A.3.1. Transport Connection Establishment Phase

Once the following sets are extracted as described in Chapter 3, they are used as input to the PSM construction algorithm.

Σ = { -TConReq@SAP_1, -TConReq@SAP_2,
+TConInd@SAP_1, +TConInd@SAP_2,
-TConRes@SAP_1, -TConRes@SAP_2,
+TConConf@SAP_1, +TConConf@SAP_2}.

I (initial events) = {-TConReq@SAP_1, -TConReq@SAP_2}

LC = { (-TConReq@SAP_1, +TConConf@SAP_1),
(-TConReq@SAP_2, +TConConf@SAP_2),
(+TConInd@SAP_2, -TConRes@SAP_2),
(+TConInd@SAP_1, -TConRes@SAP_1) }.

E2EC={ (-TConReq@SAP_1, +TConInd@SAP_2),
(-TConRes@SAP_2, +TConConf@SAP_1),
(-TConReq@SAP_2, +TConInd@SAP_1),
(-TConRes@SAP_1, +TConConf@SAP_2) }

$CC = \{\}$

The following figures depict how this phase is built by the algorithm by using the set of inputs given above. Note that this phase has a symmetric behavior. Therefore, only one side of the phase is shown. In Figure 41, the completed PSM for the Transport Connection Establishment Phase is shown.

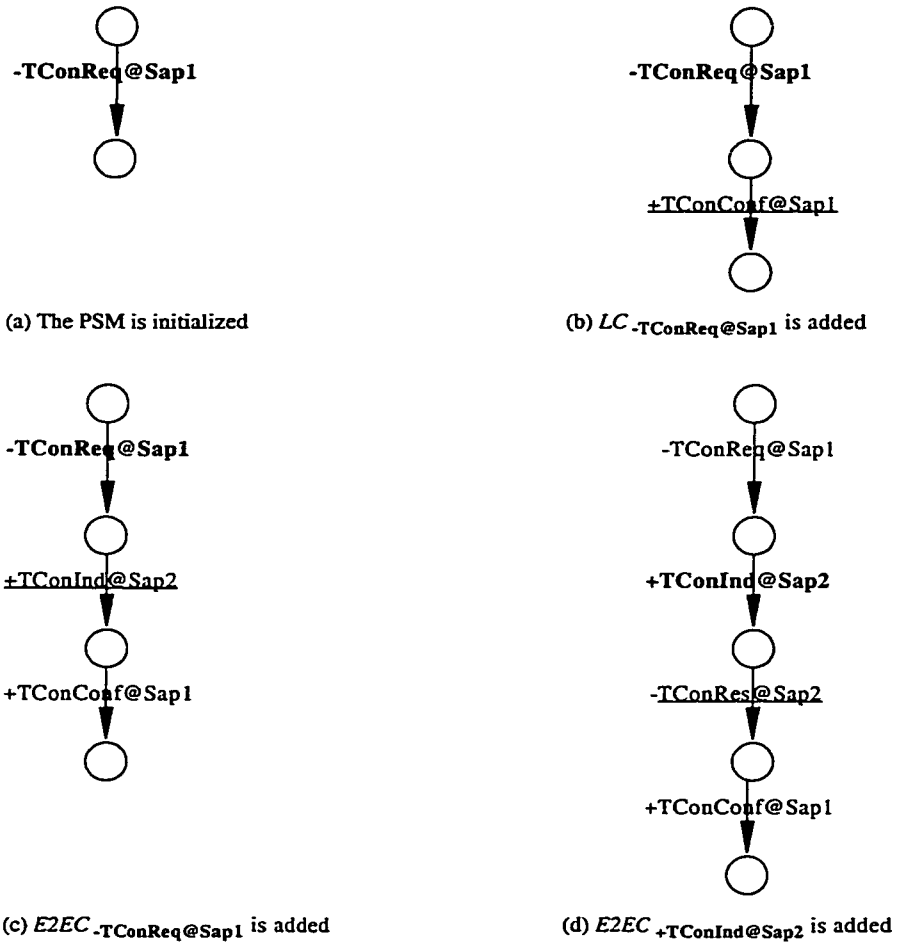


Figure 40. Construction of the PSM for Transport Connection Establishment Phase (Only one half is shown)

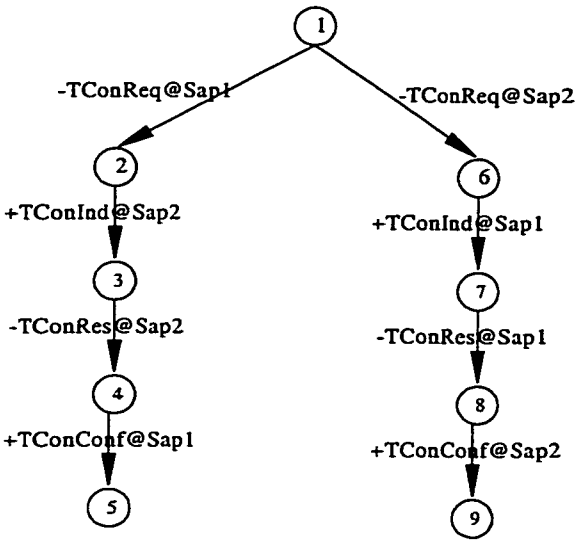


Figure 41. The Transport Connection Establishment Phase

A.3.2. Transport Connection Data Transfer Phase

The following sets are derived for this phase and will be used by the algorithm to build the PSM for Data Transfer Phase:

Σ = { -TDataReq@SAP$_1$, -TDataReq@SAP$_2$,
+TDataInd@SAP$_1$, +TDataInd@SAP$_2$ }.

I (initial events) ={ -TDataReq@SAP$_1$, -TDataReq@SAP$_2$}

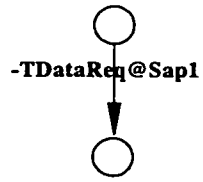
LC = { (-TDataReq@SAP$_1$, -TDataReq@SAP$_1$),
(-TDataReq@SAP$_2$, -TDataReq@SAP$_2$),
(-TDataReq@SAP$_1$, +TDataInd@SAP$_1$),
(-TDataReq@SAP$_2$, +TDataInd@SAP$_2$),
(+TDataInd@SAP$_1$, +TDataInd@SAP$_1$),
(+TDataInd@SAP$_2$, +TDataInd@SAP$_2$),
(+TDataInd@SAP$_1$, -TDataReq@SAP$_1$),
(+TDataInd@SAP$_2$, -TDataReq@SAP$_2$) }.

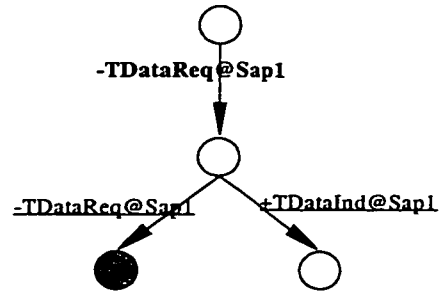
E2EC={ (-TDataReq@SAP$_1$, +TDataInd@SAP$_2$),
(-TDataReq@SAP$_2$, +TDataInd@SAP$_1$)}

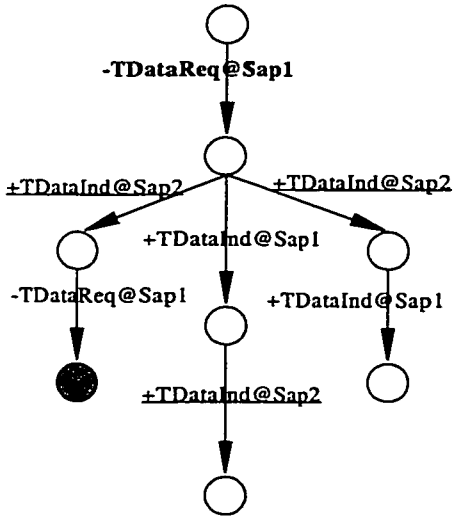
CC = { (-TDataReq@SAP$_1$,-TDataReq@SAP$_2$),
(-TDataReq@SAP$_2$,-TDataReq@SAP$_1$),
(+TDataInd@SAP$_1$,+TDataInd@SAP$_2$),
(+TDataInd@SAP$_2$,+TDataInd@SAP$_1$)}

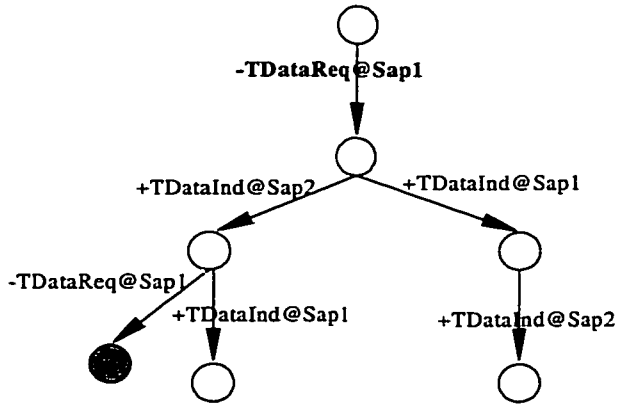
The following series of figures are the execution of the algorithm step by step.

Pass 1:

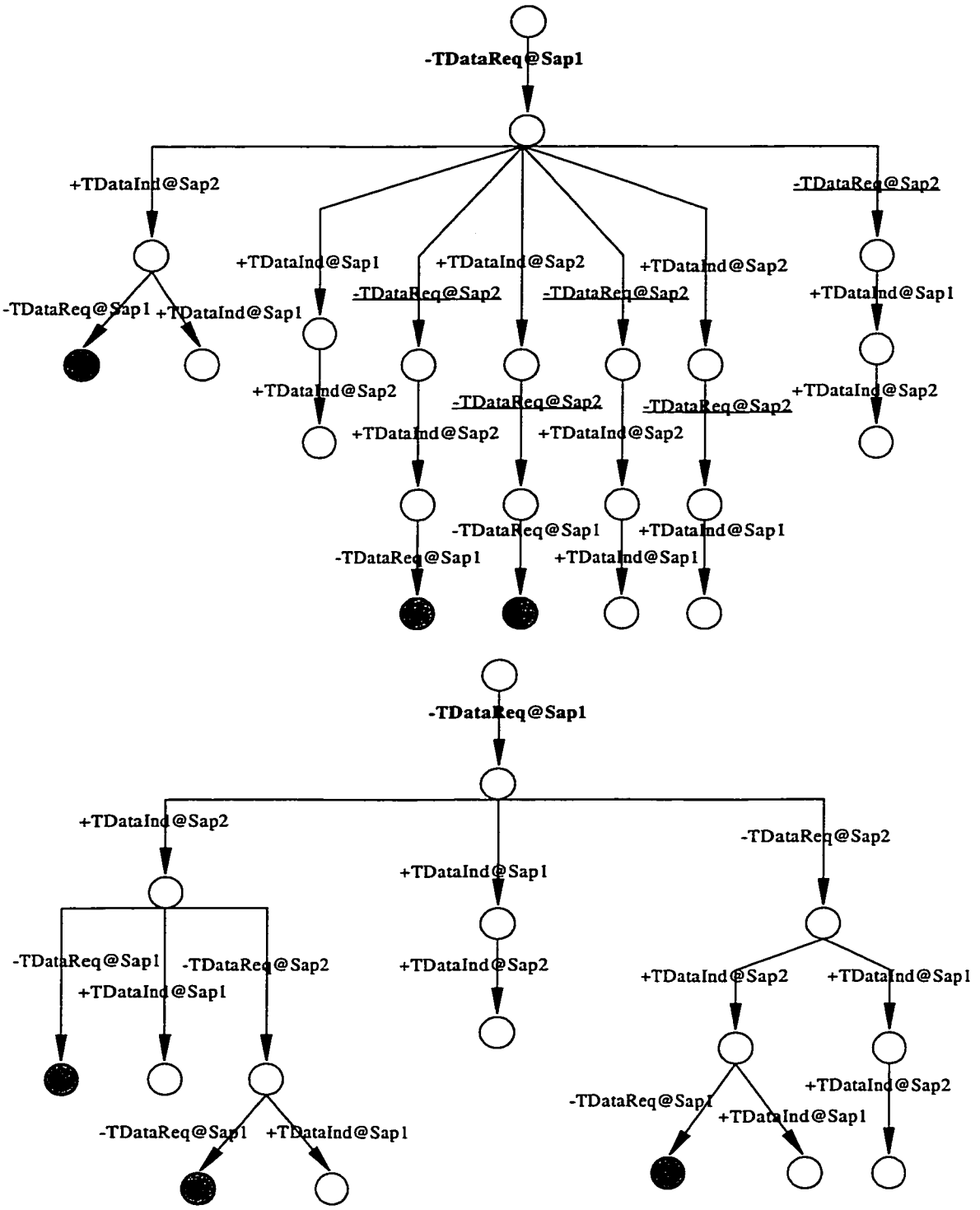






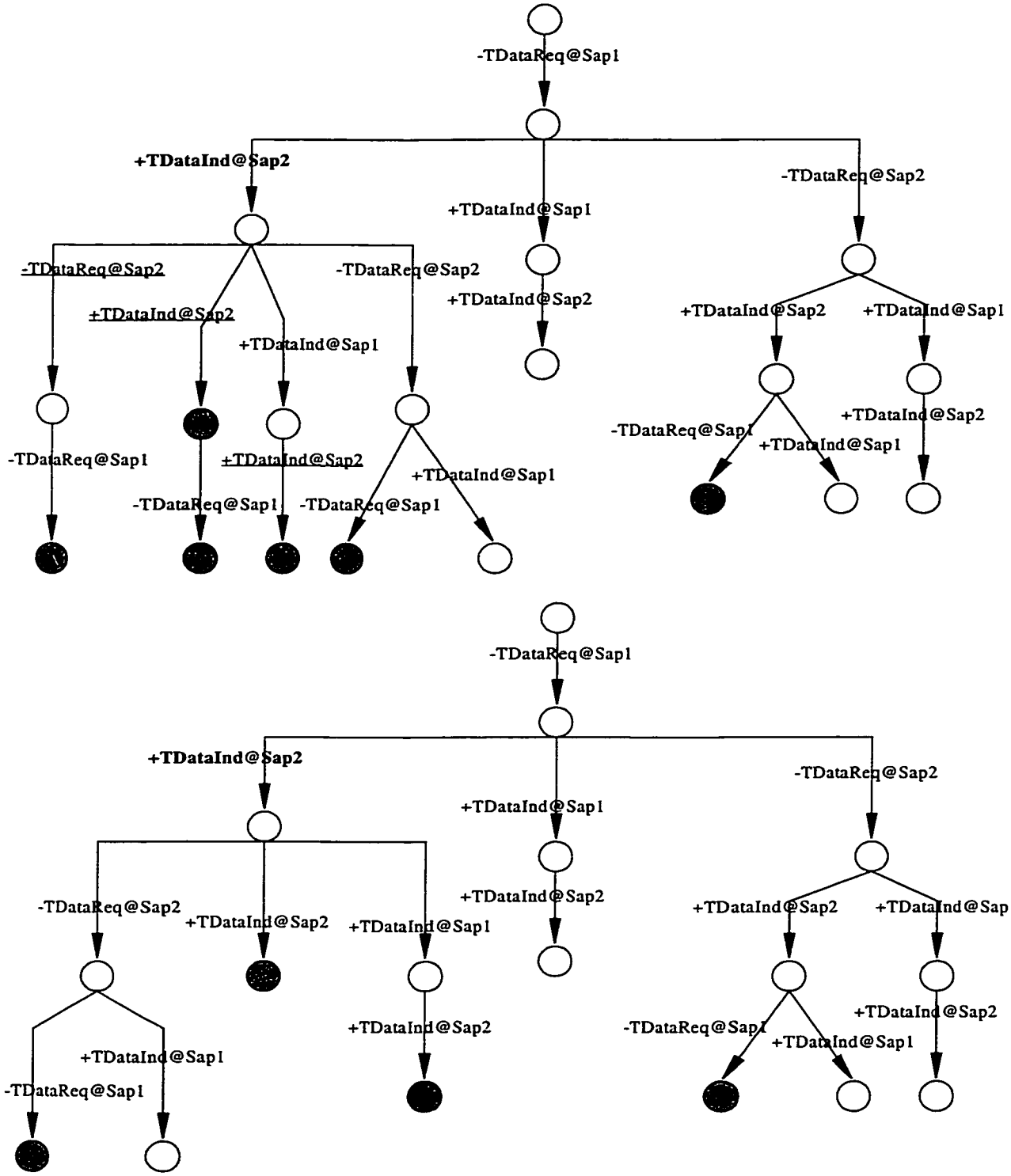


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*



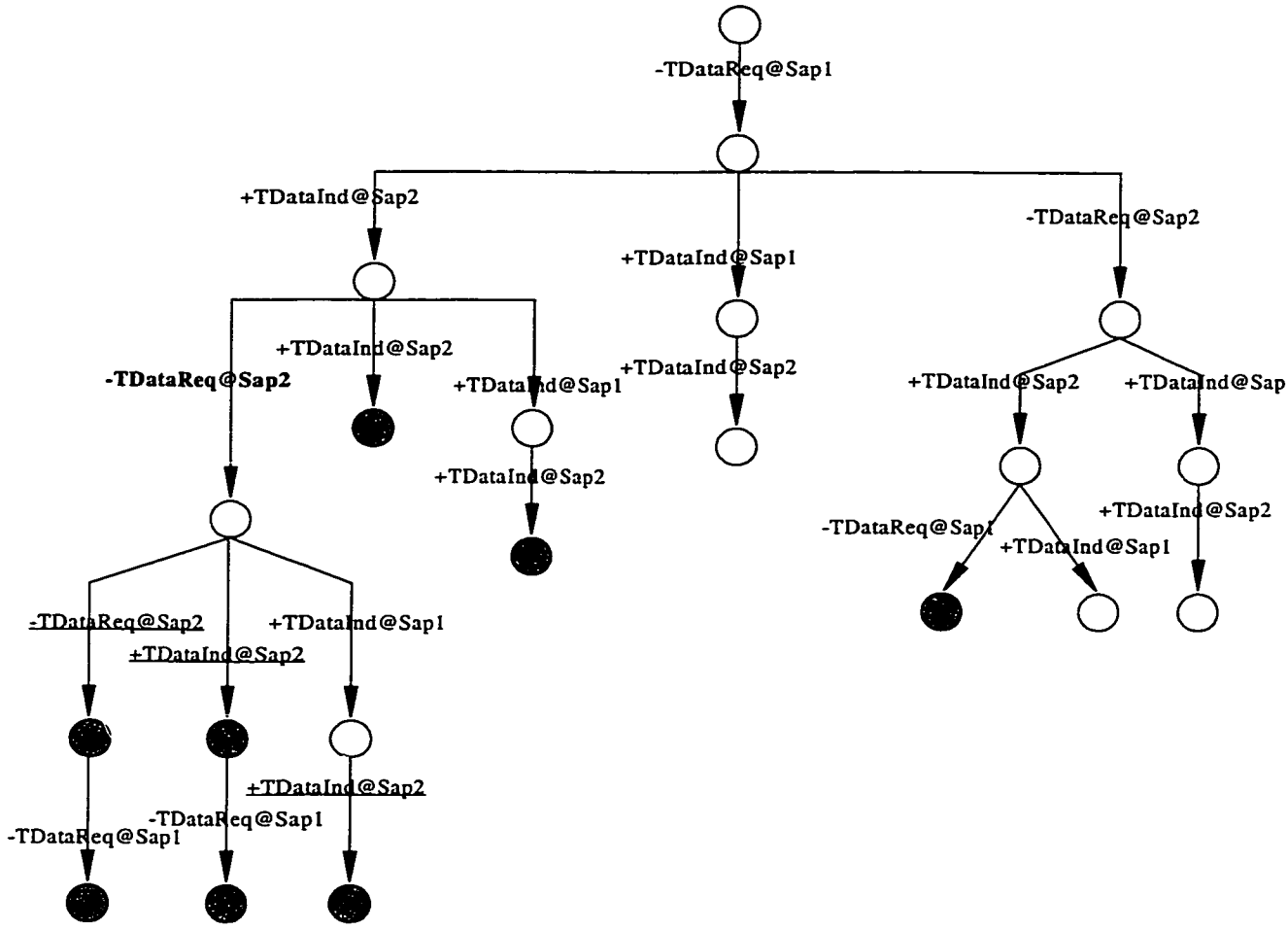
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 2:

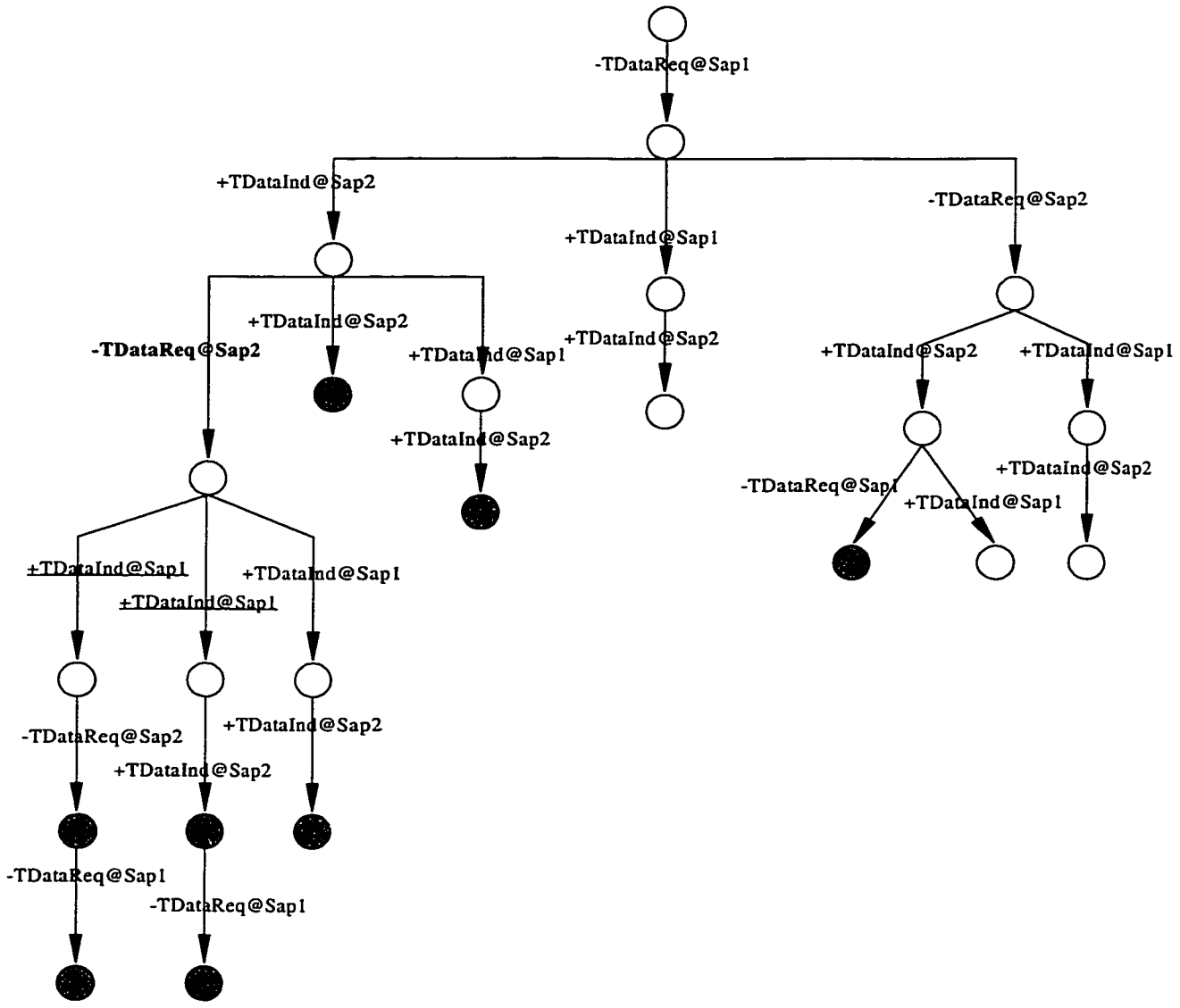


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

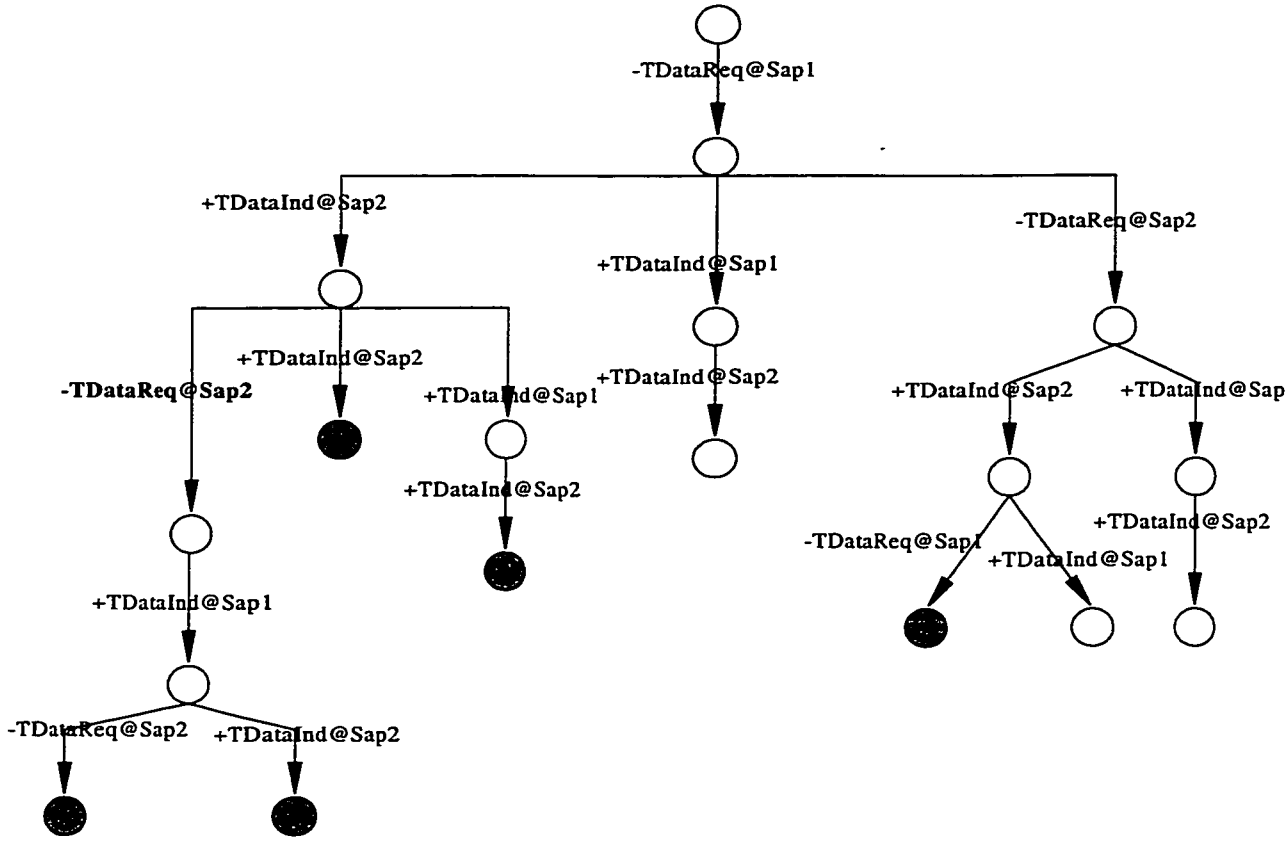
Pass 3:



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

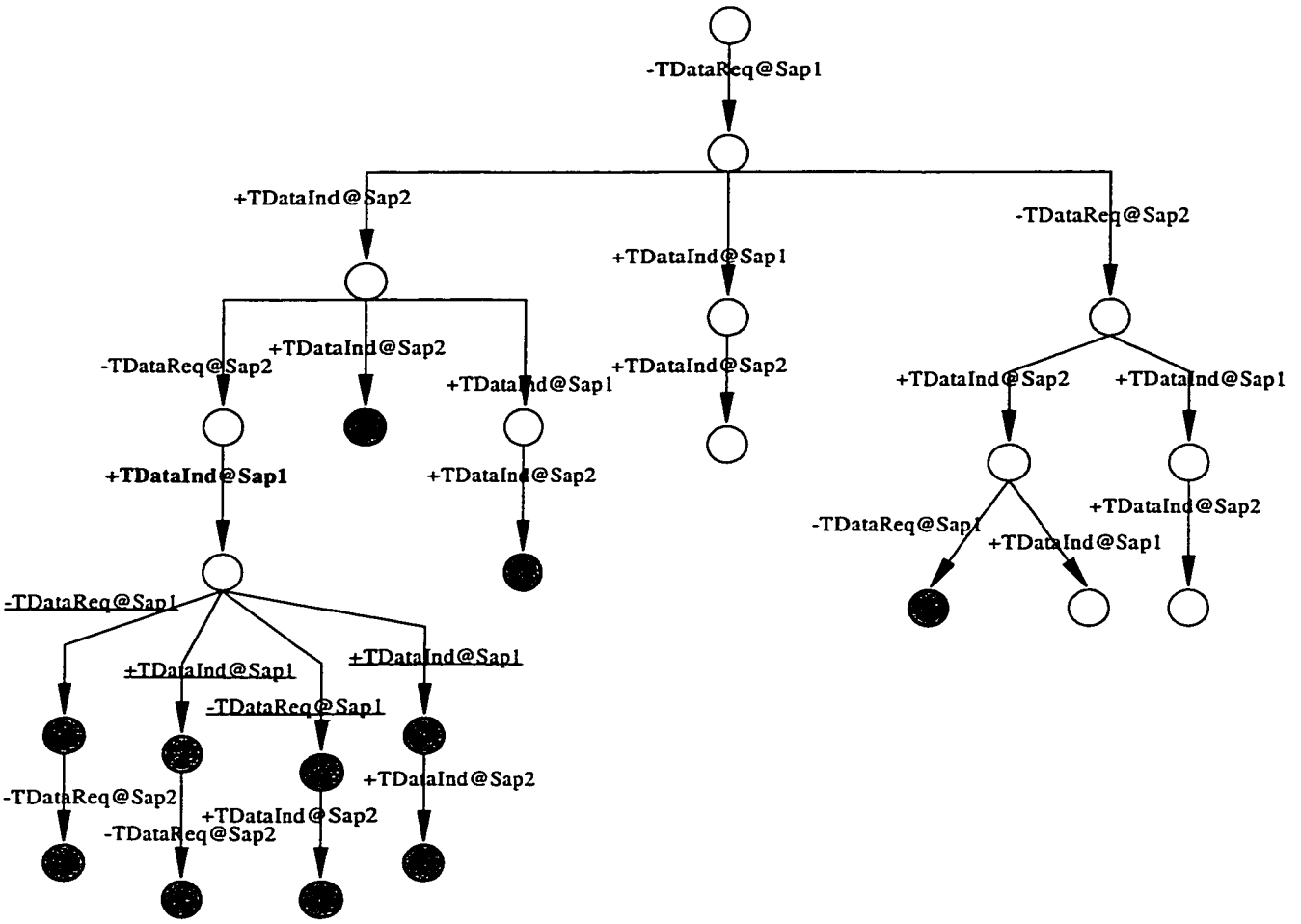


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

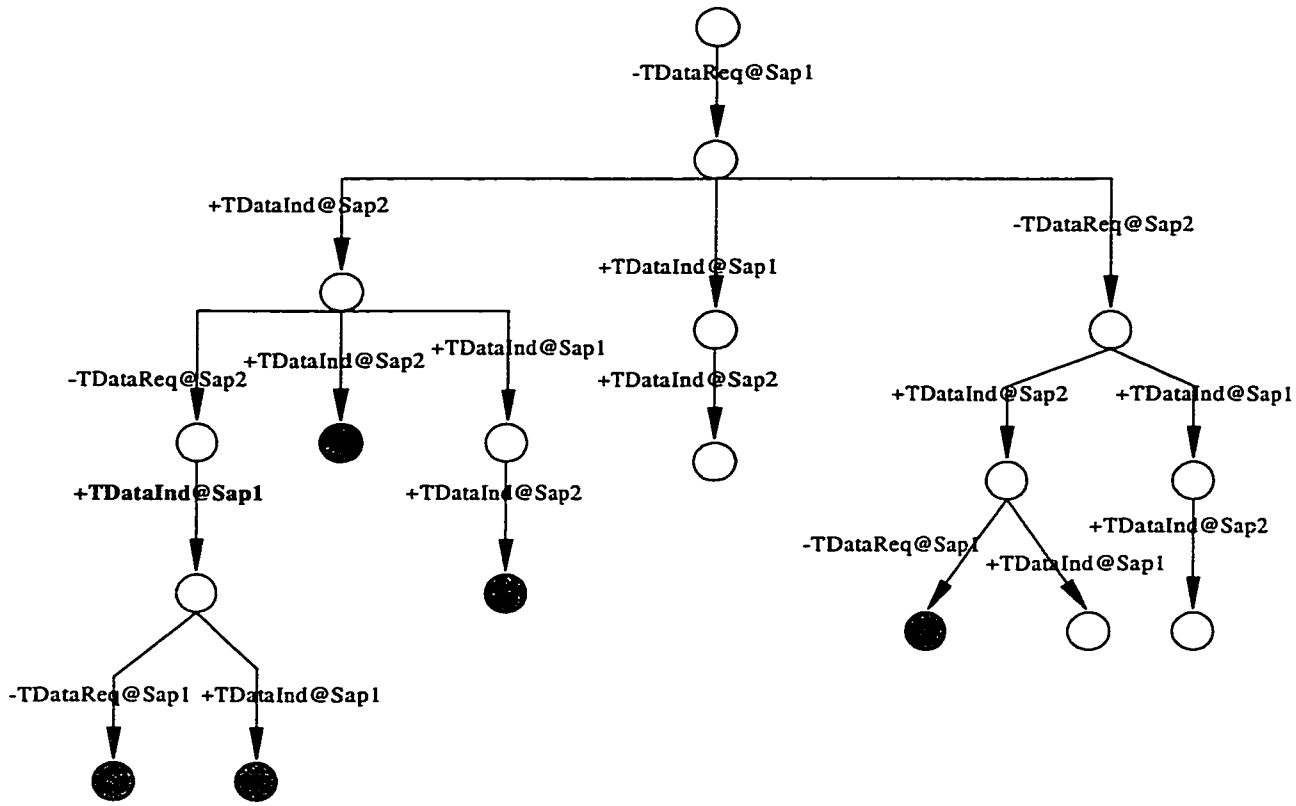


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 4:

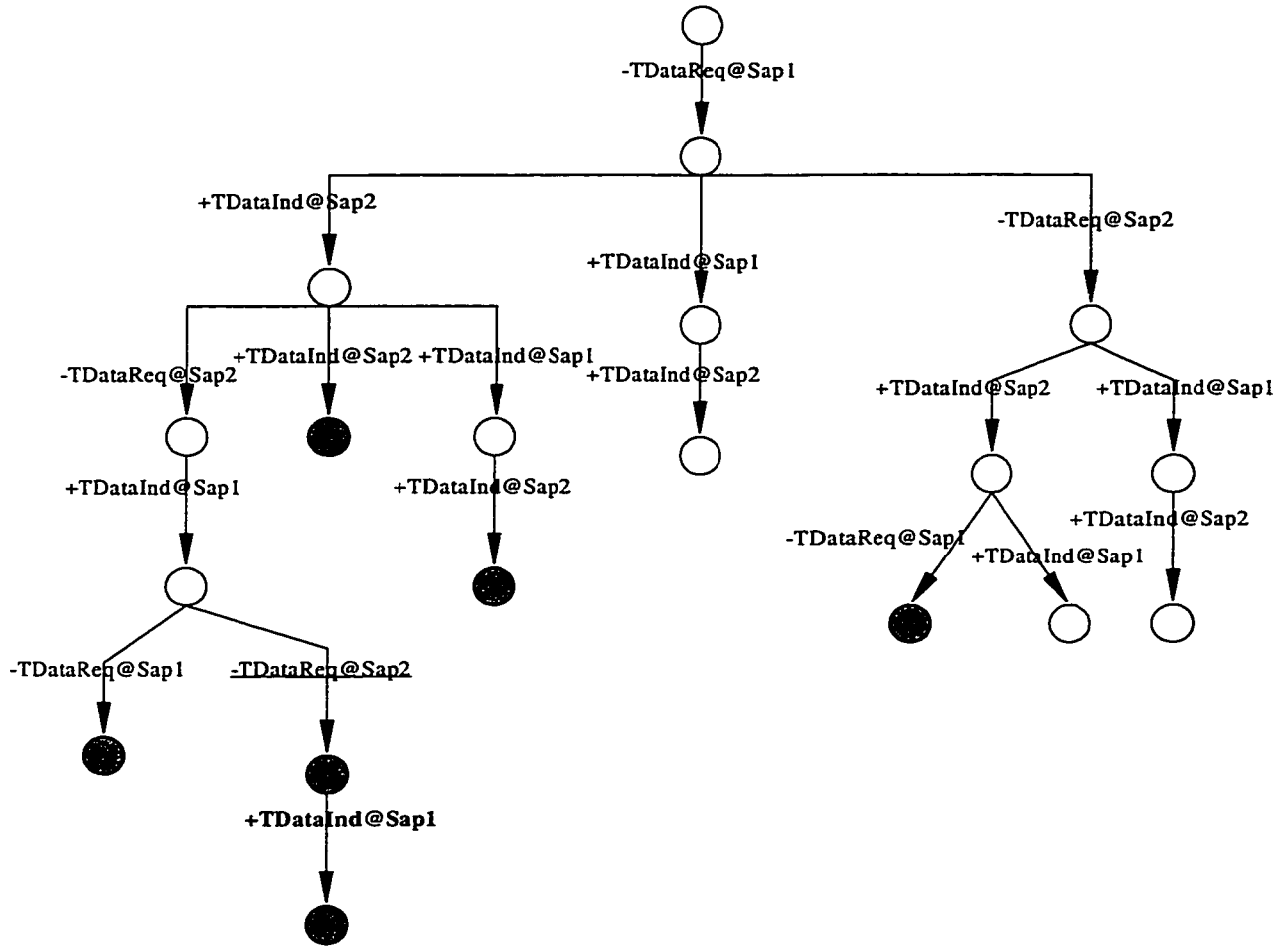


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

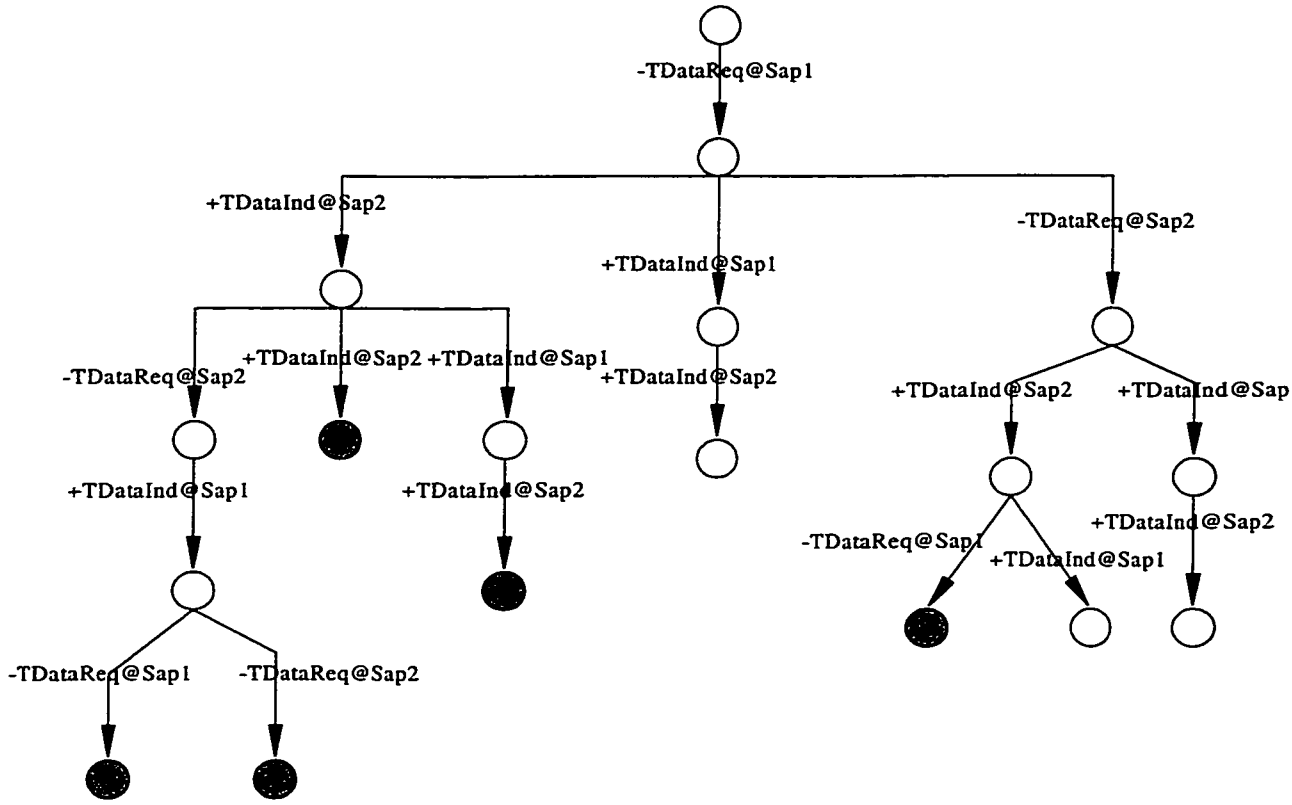


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 5:

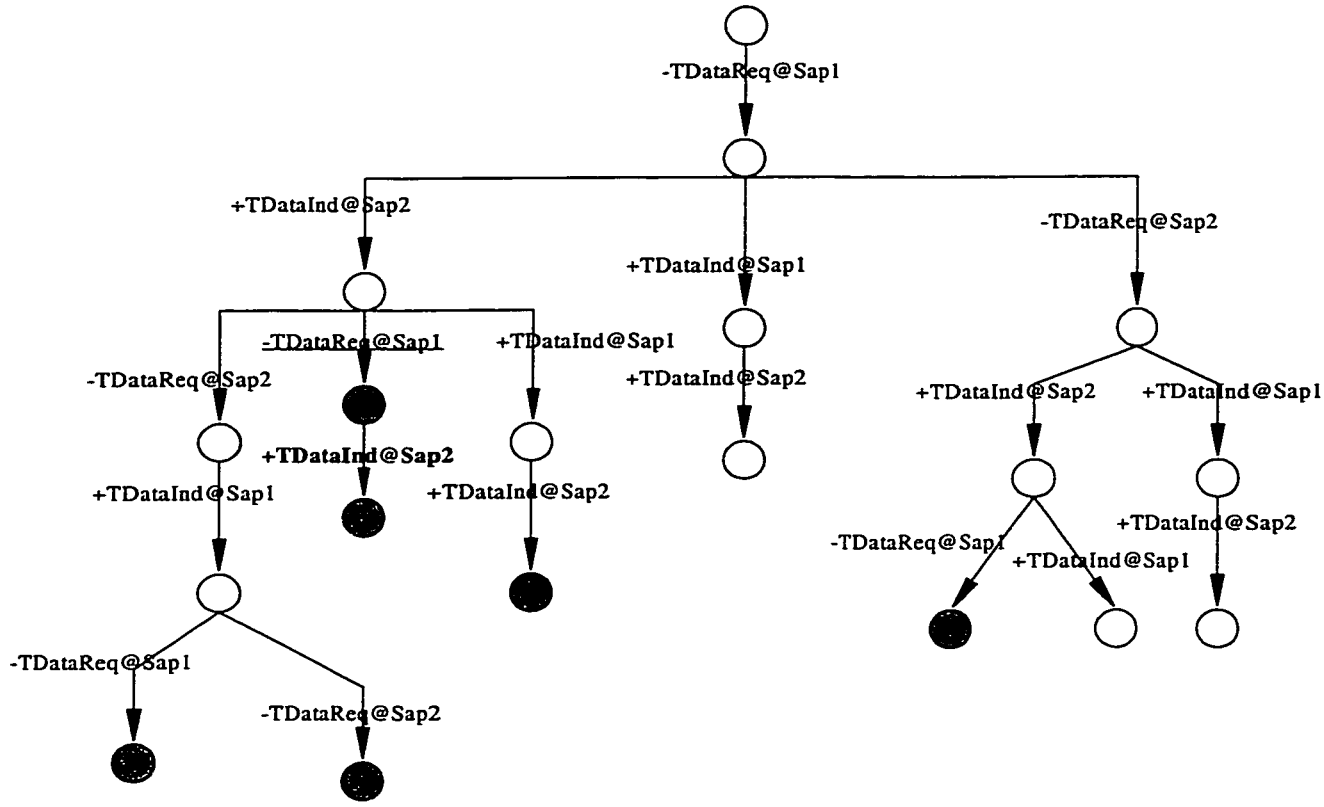


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

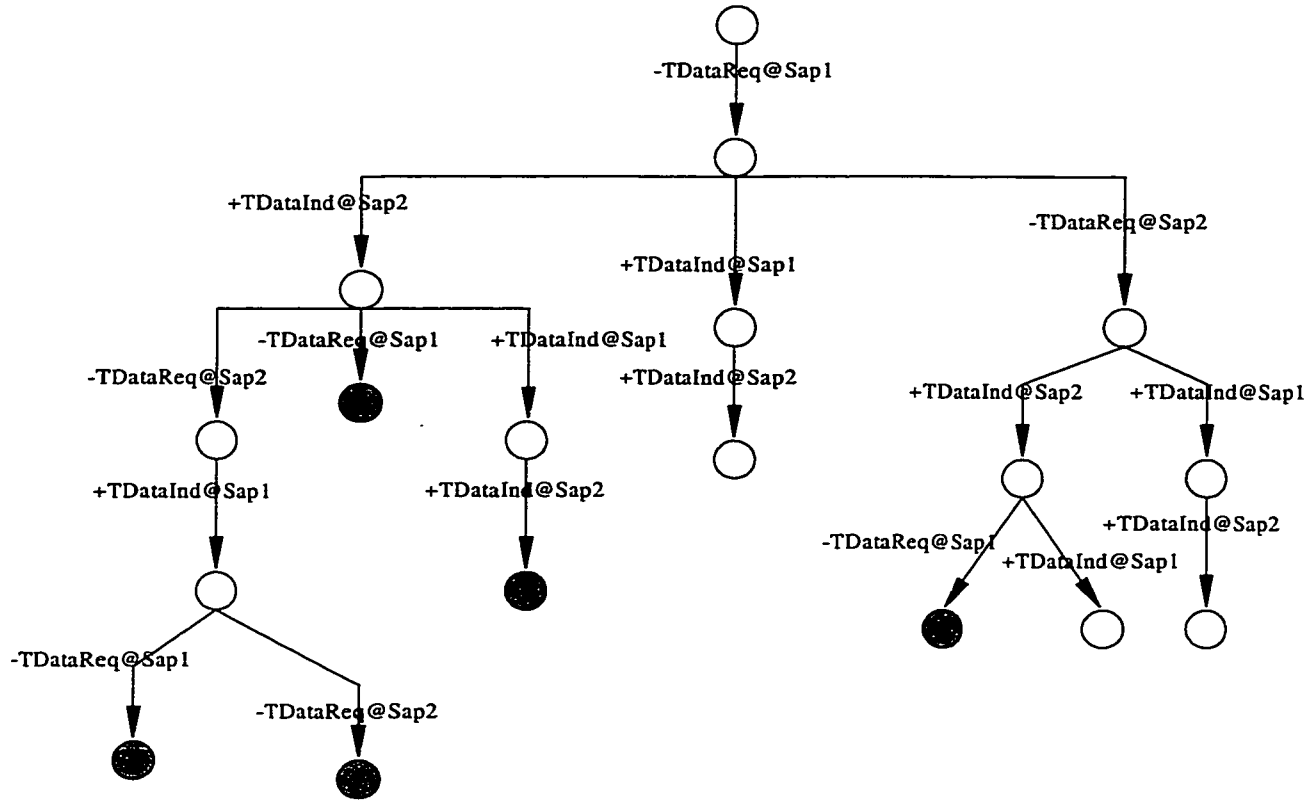


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 6:

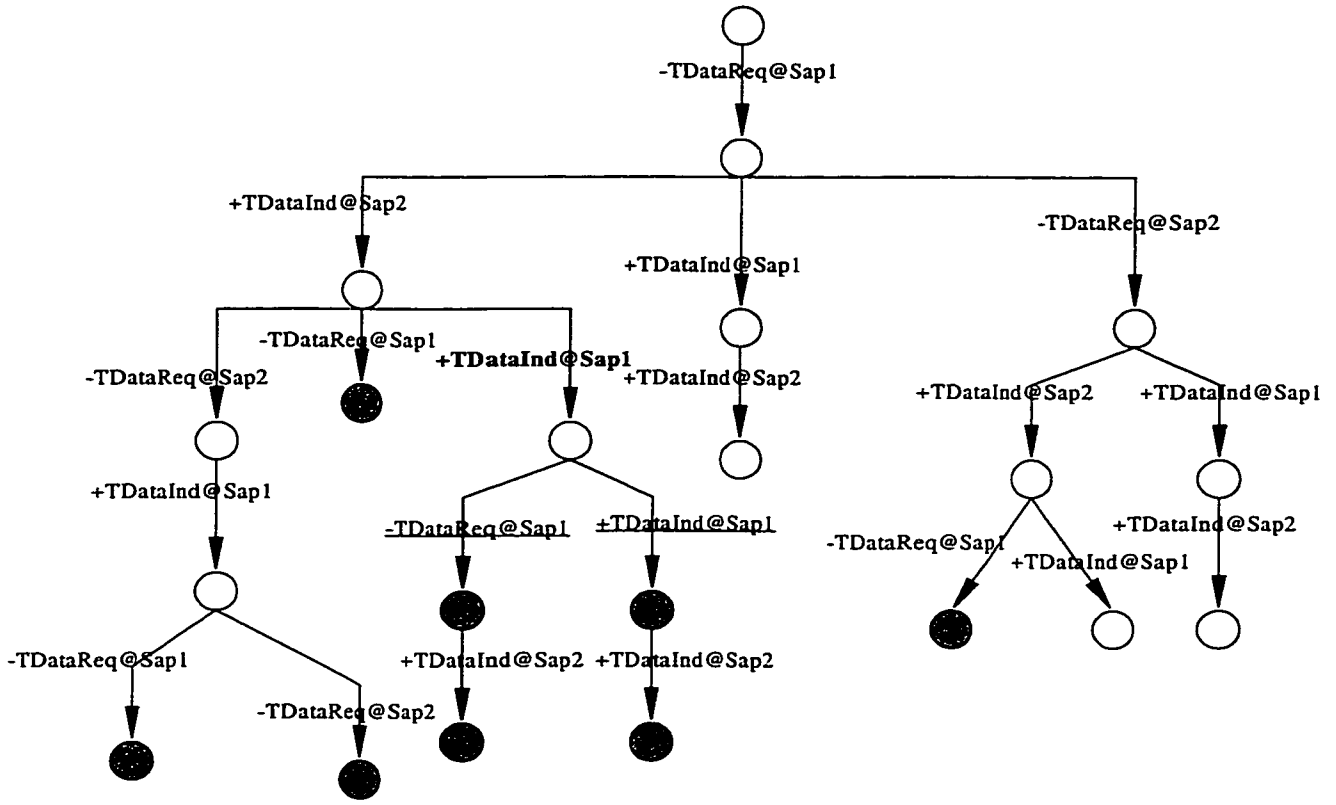


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

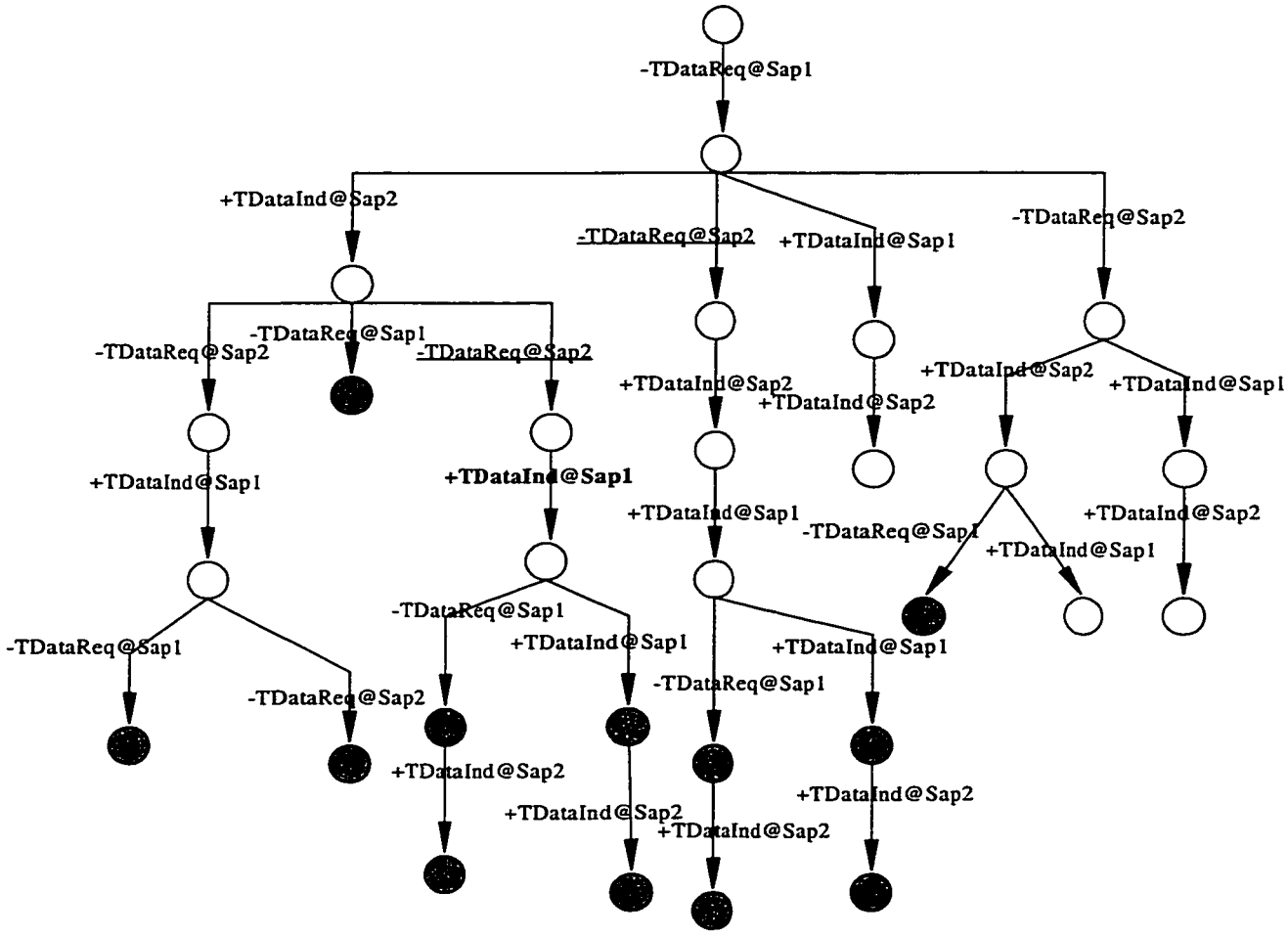


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

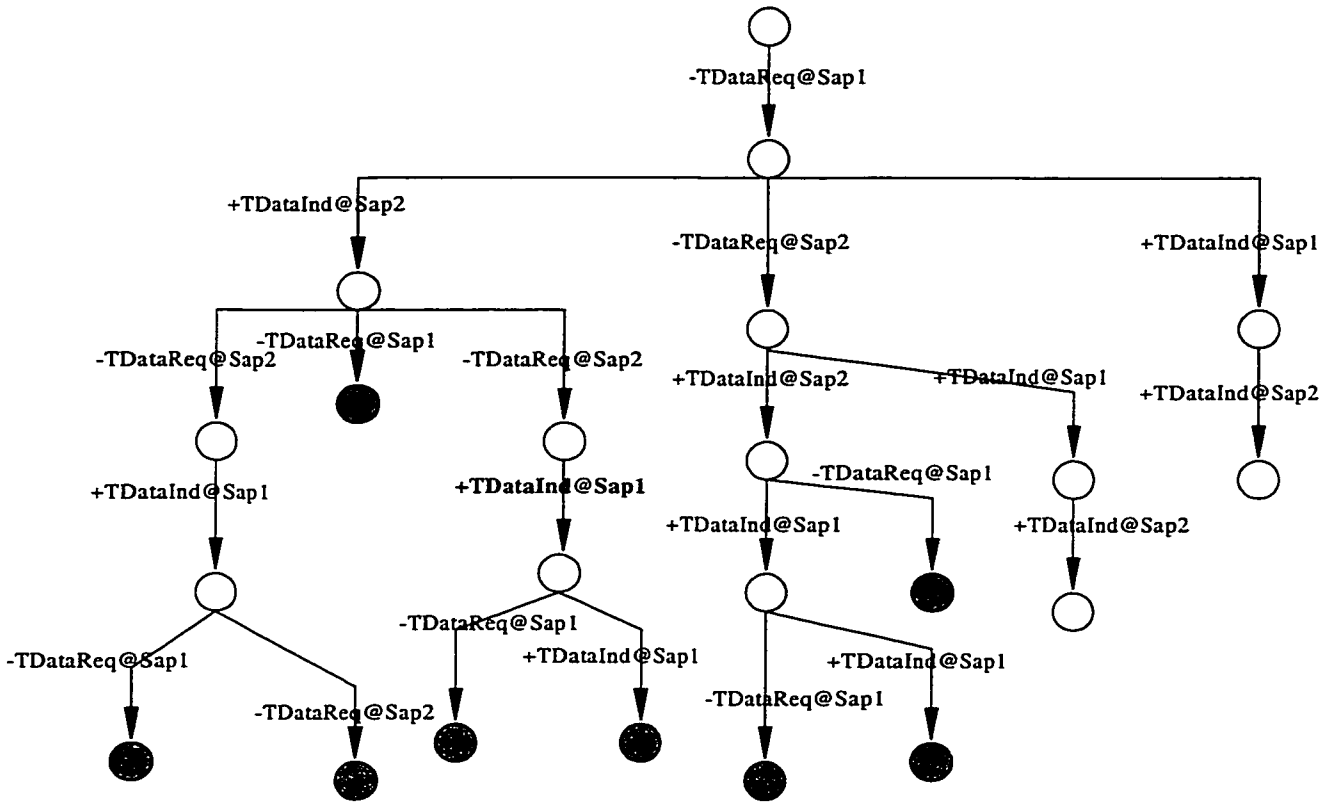
Pass 7:



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

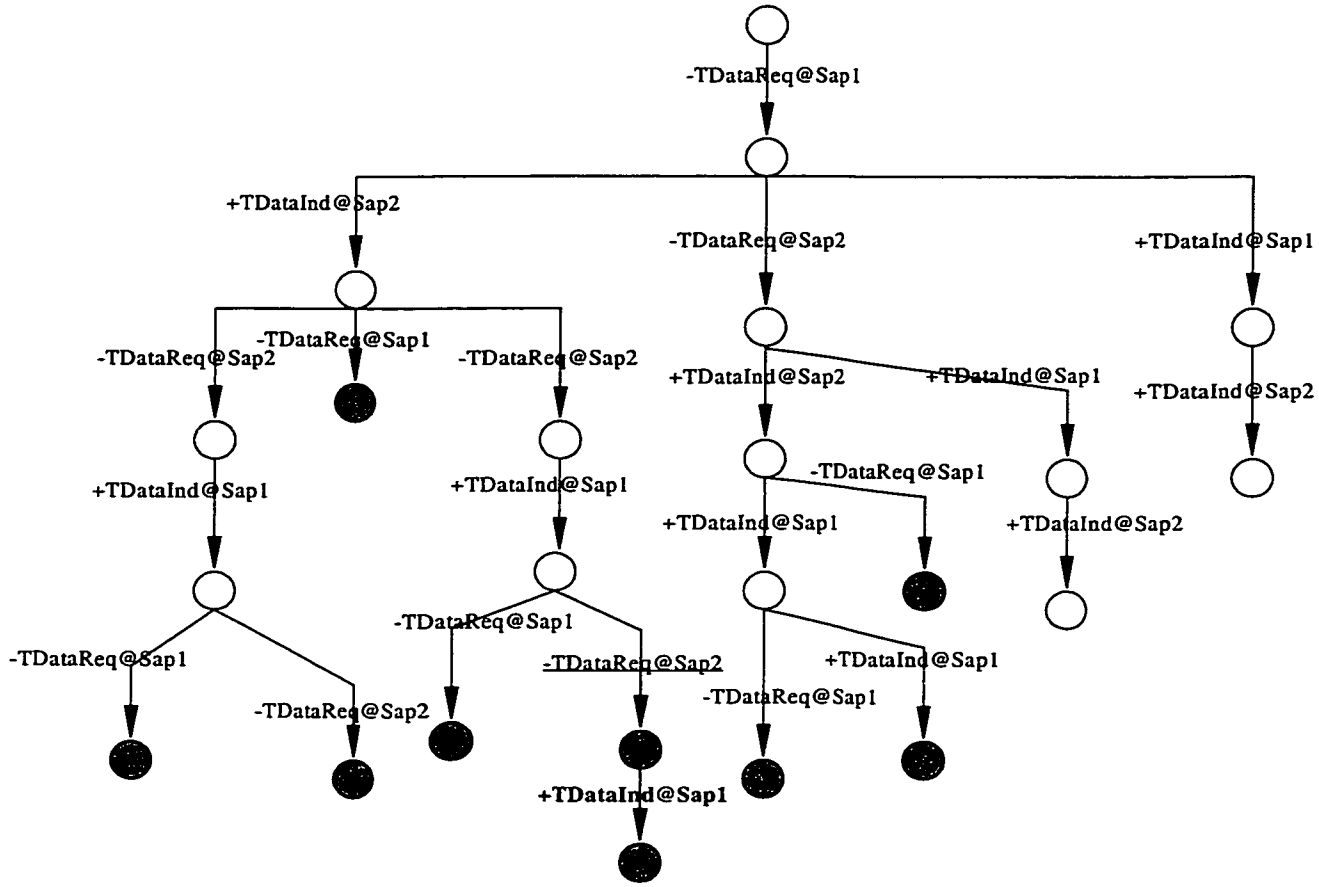


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*



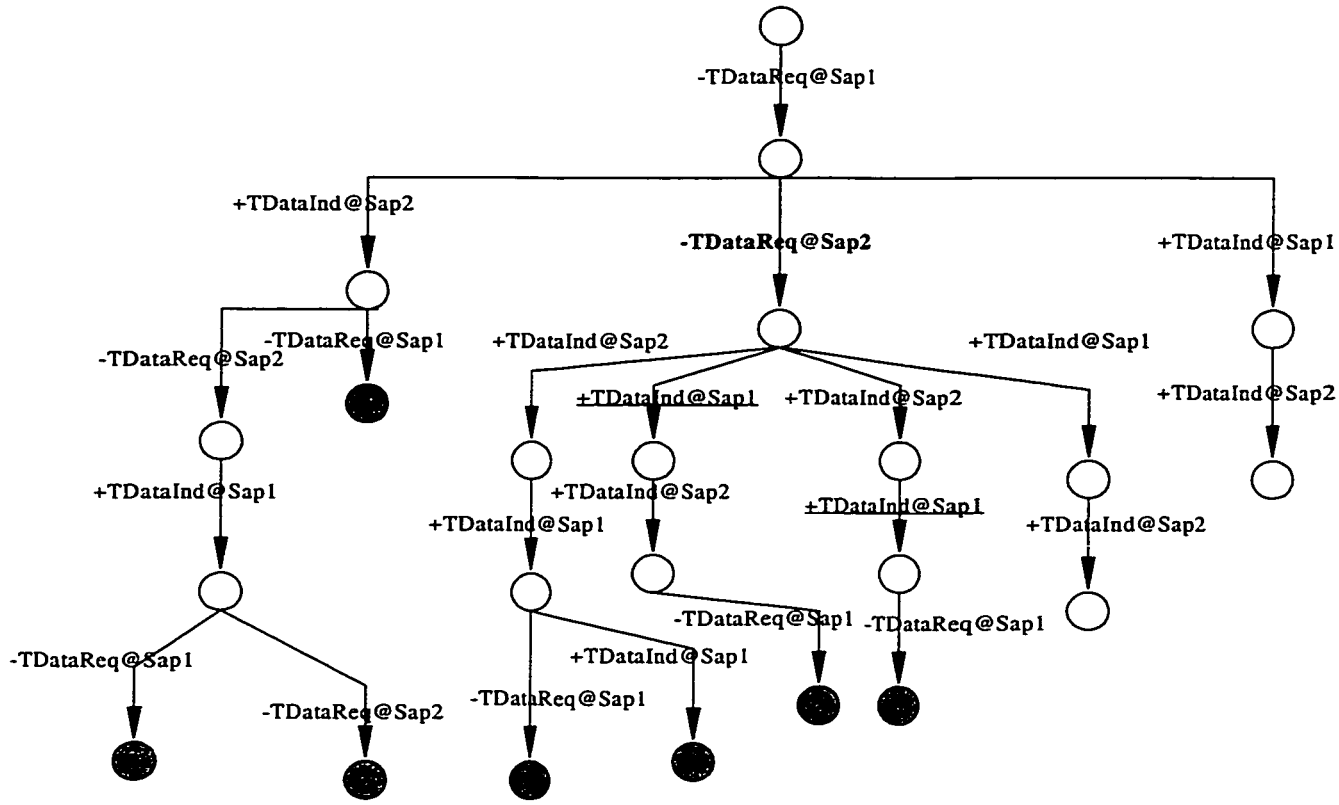
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 8:

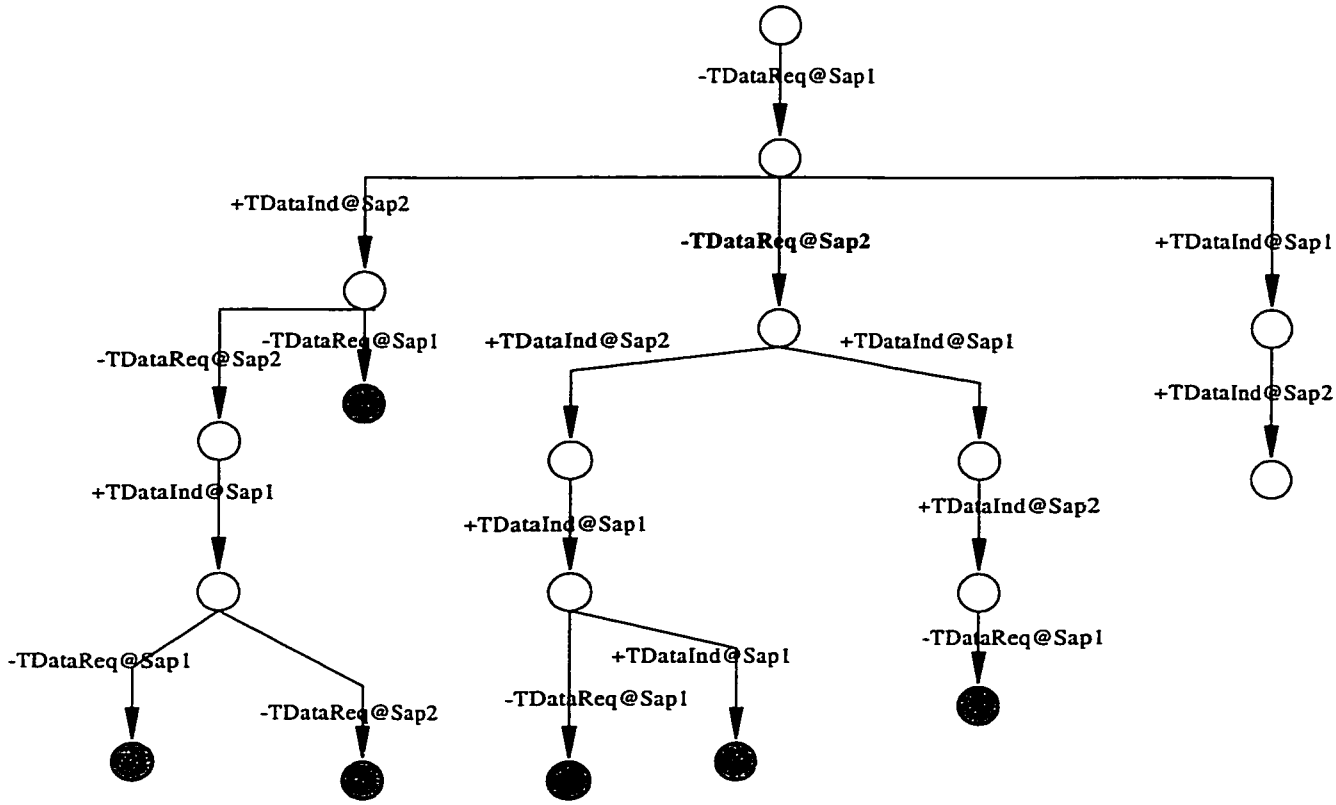


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 9:

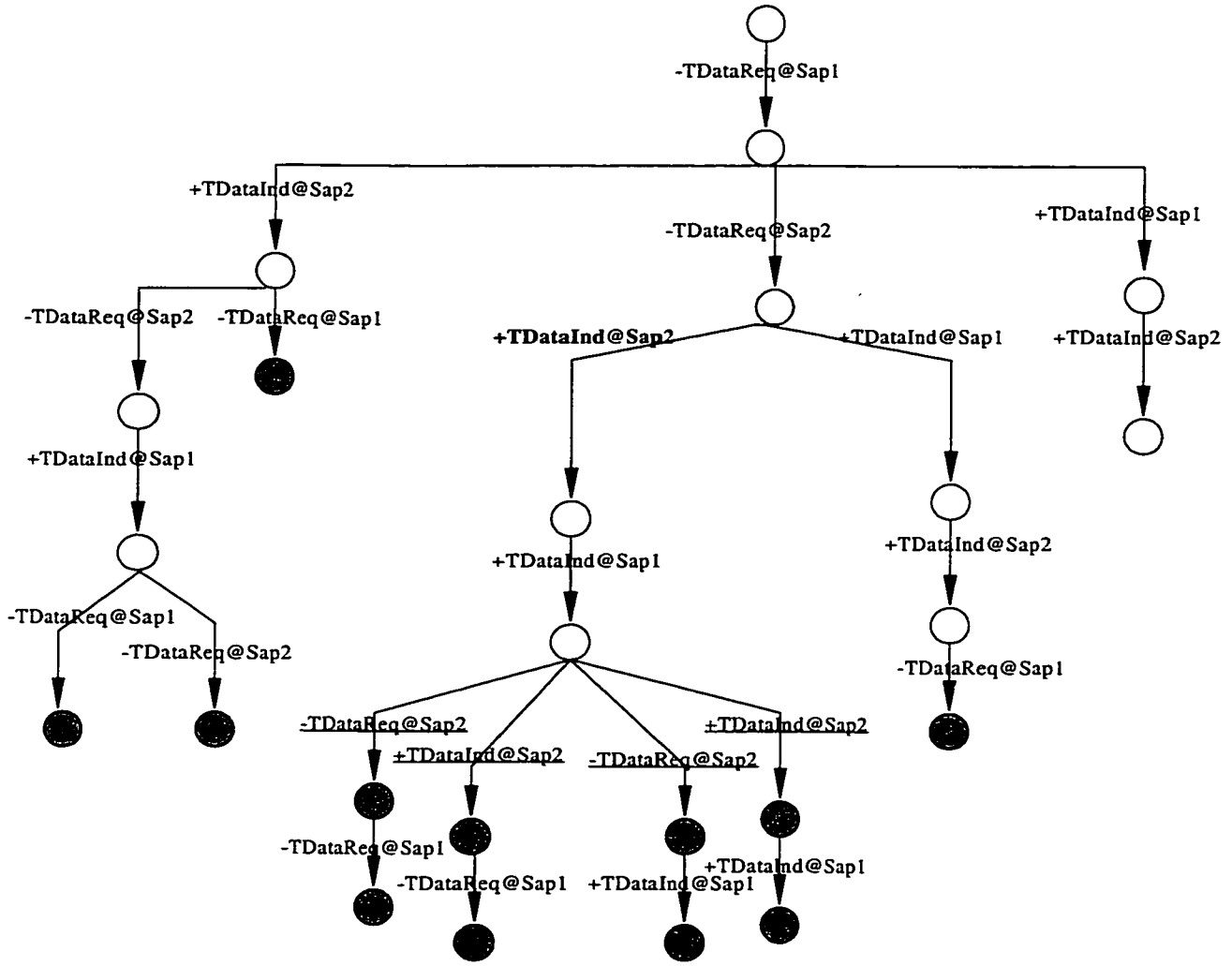


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

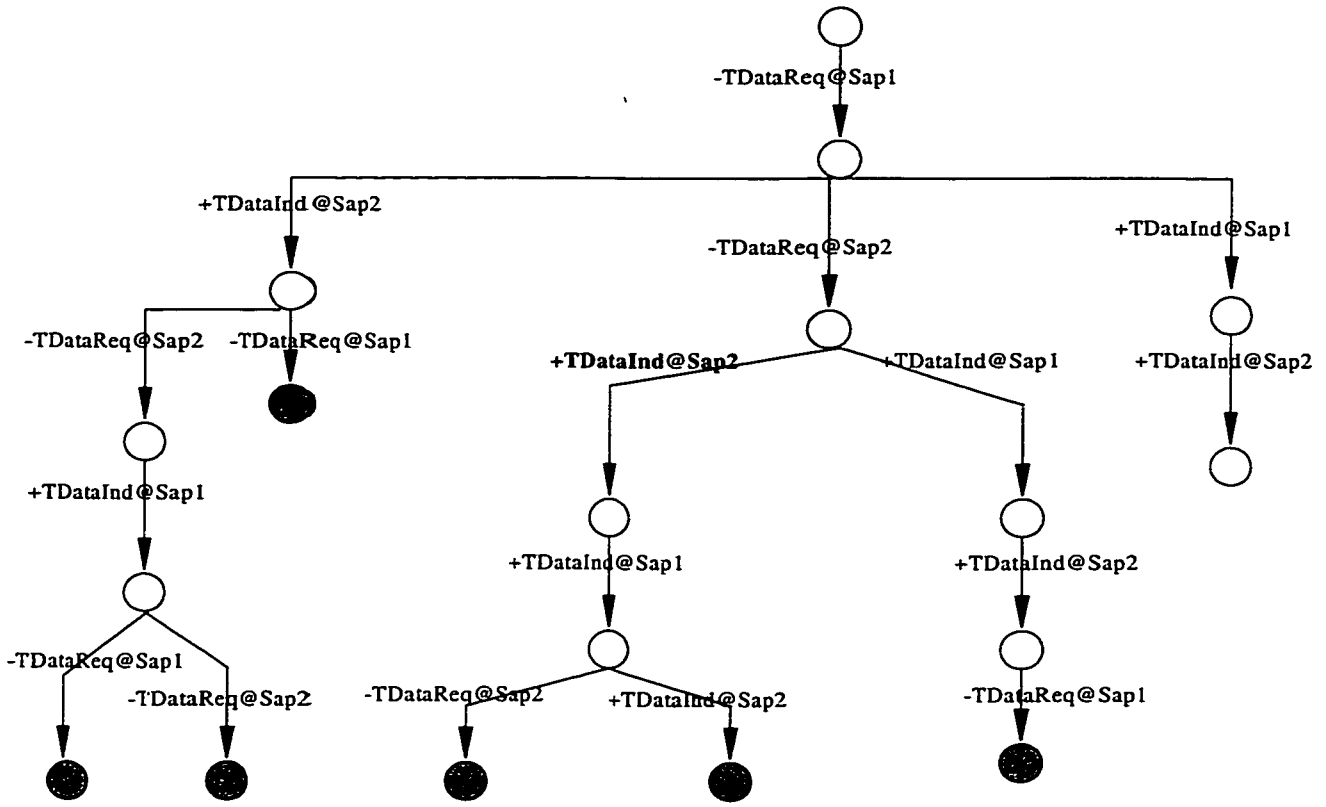


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 10:

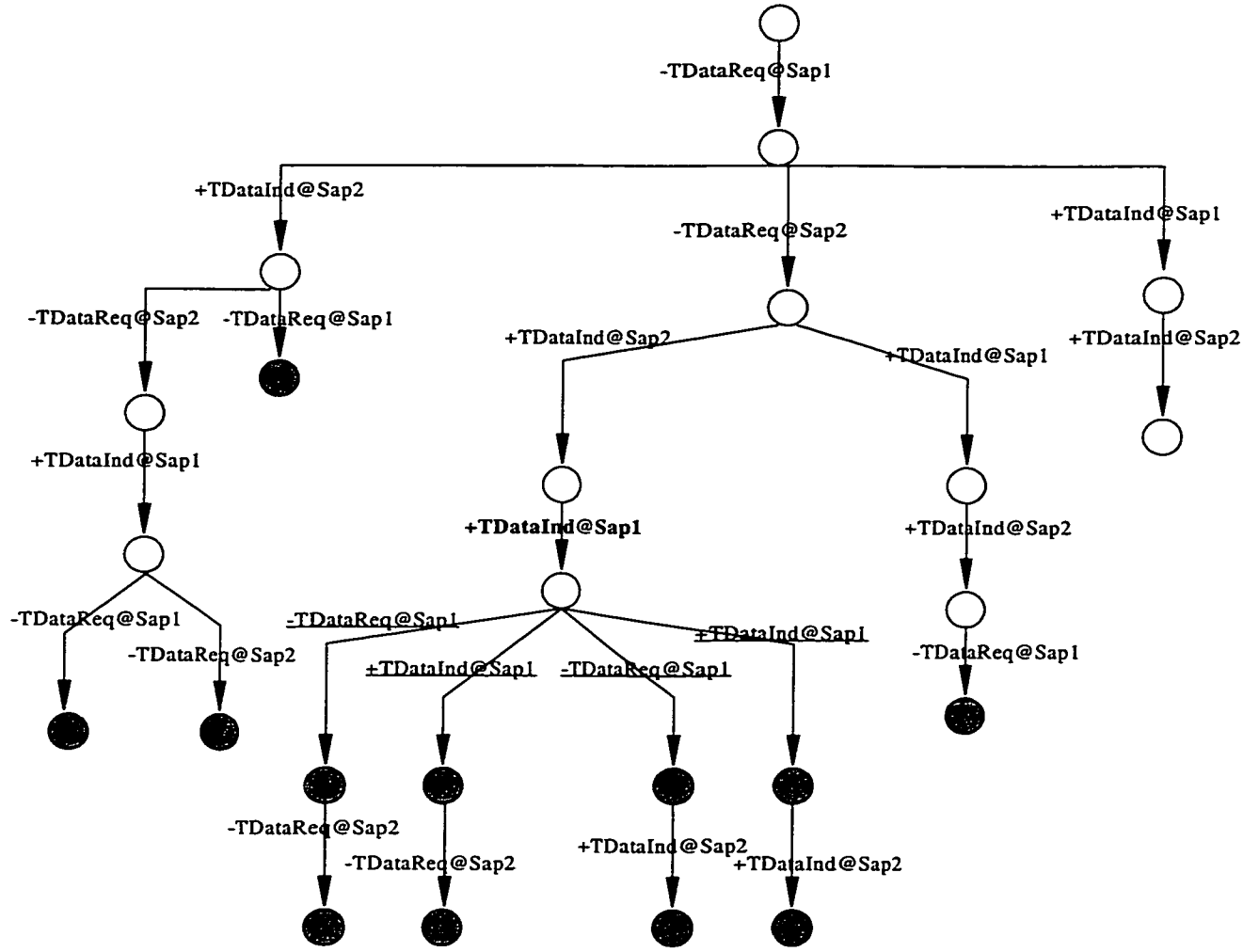


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

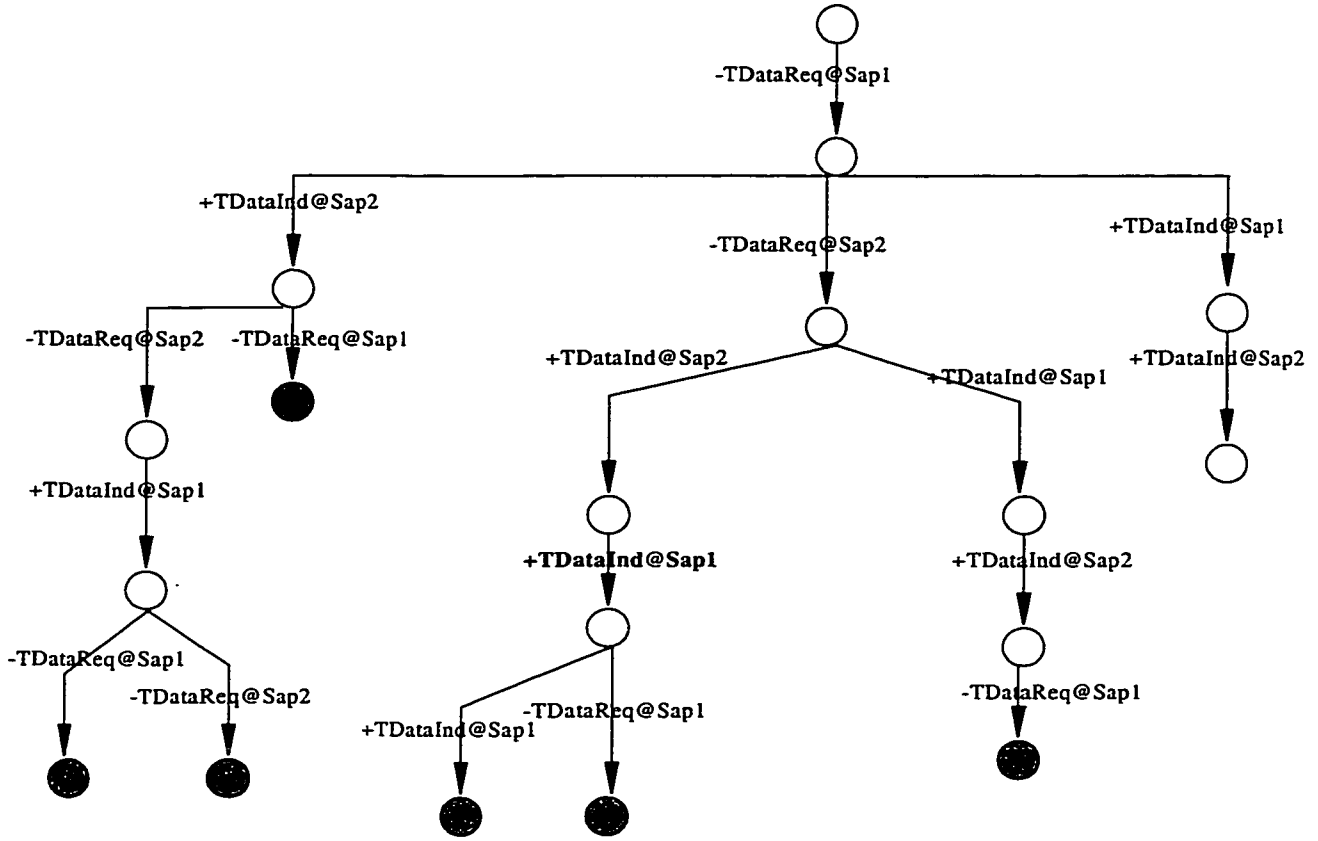


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 11:

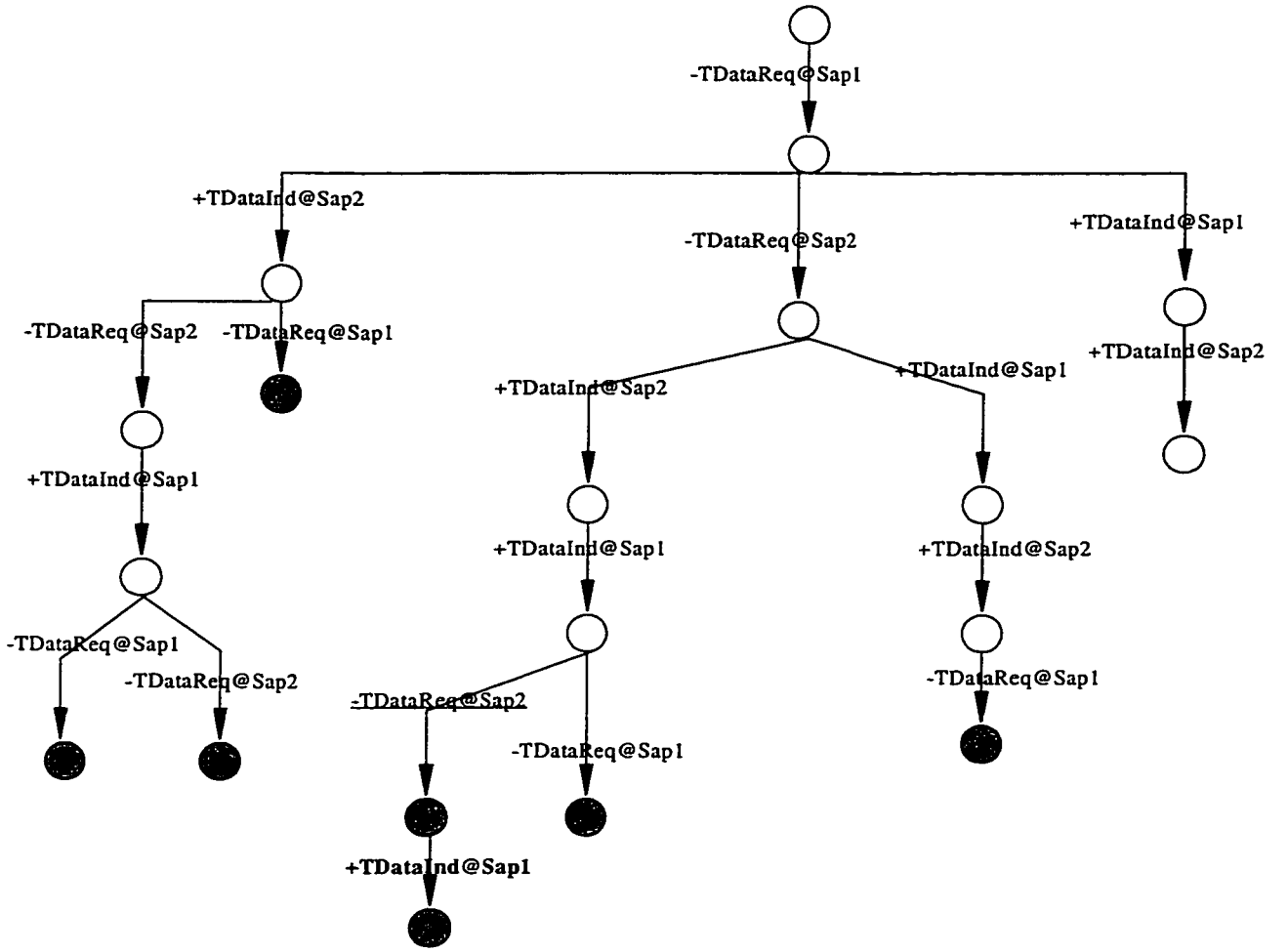


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*



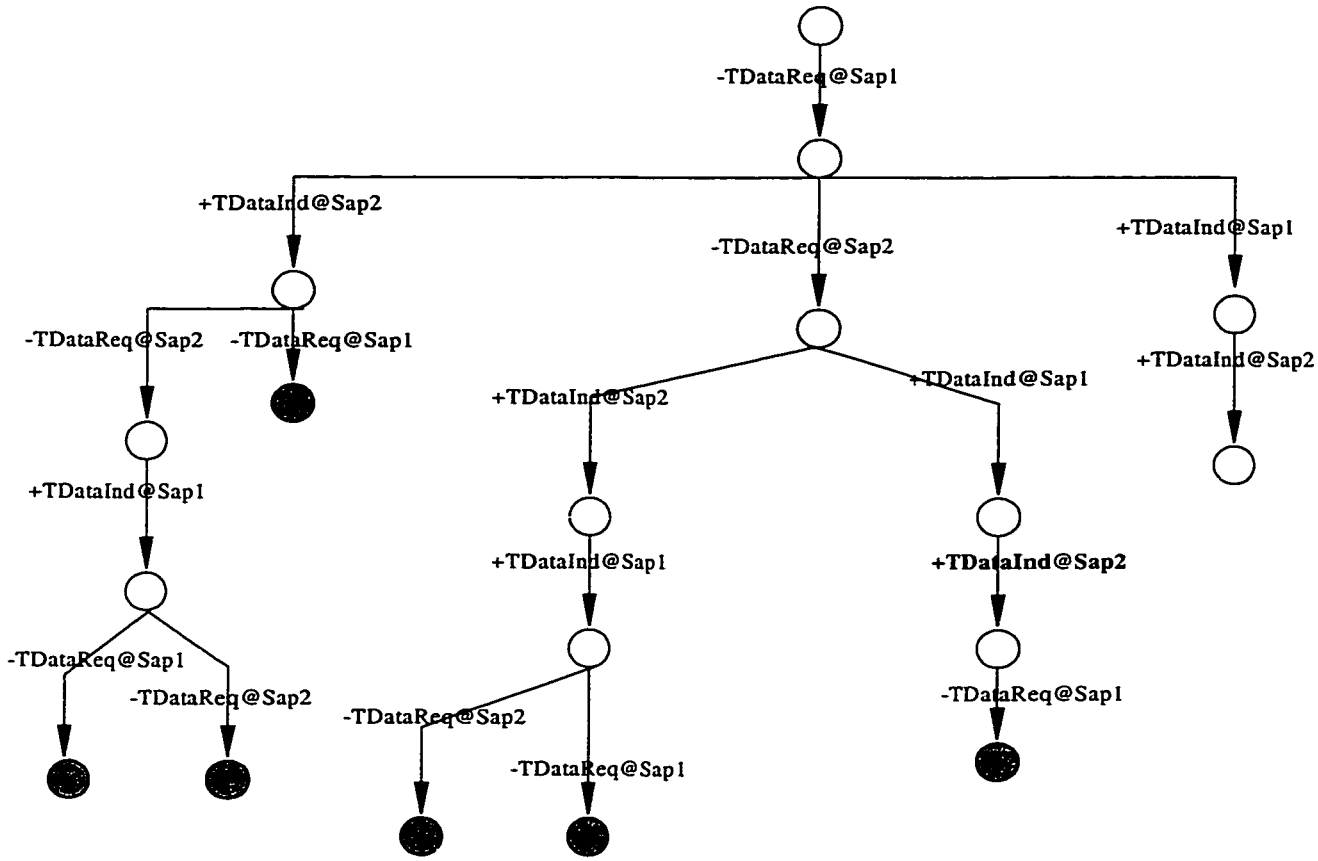
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 12:

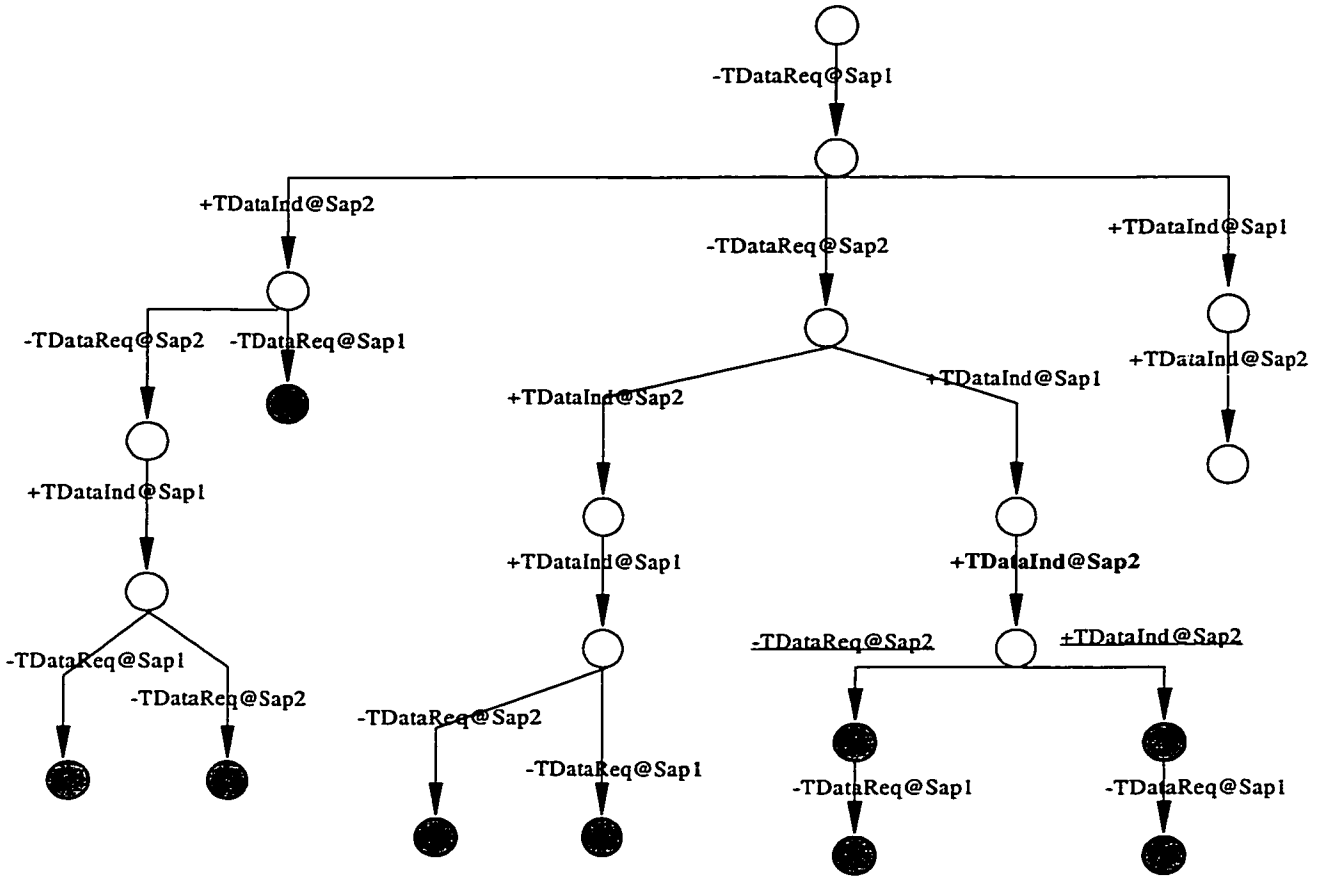


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 13:

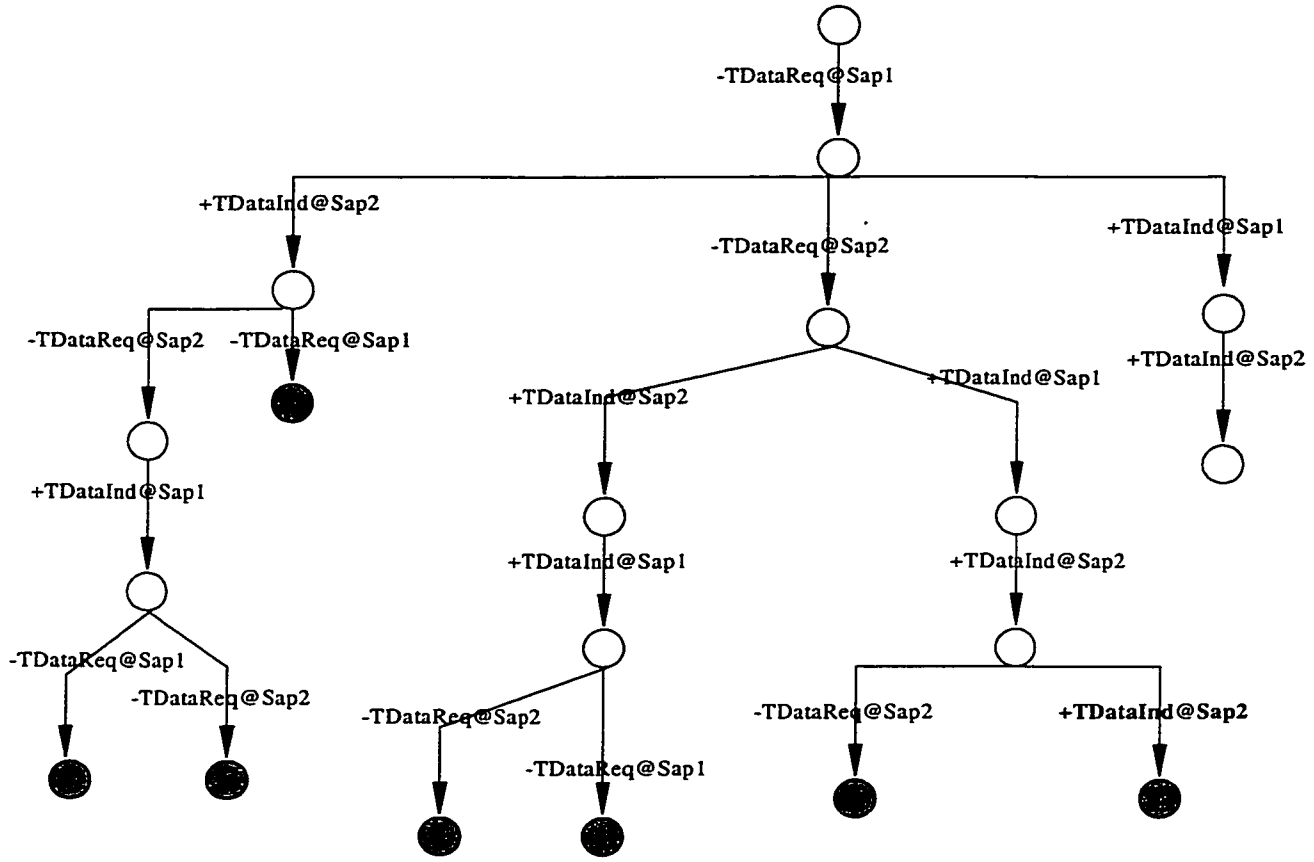


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

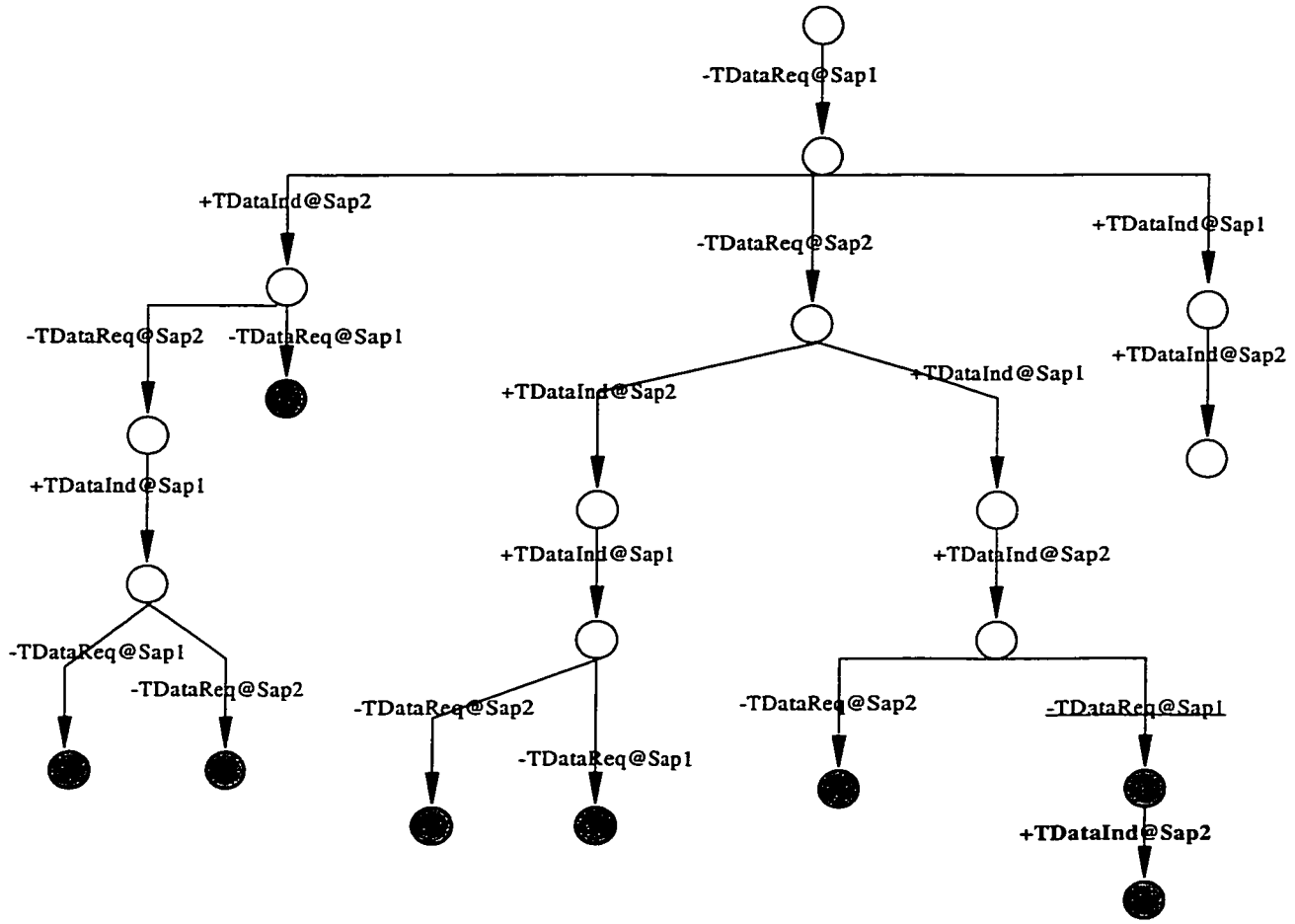


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 14:

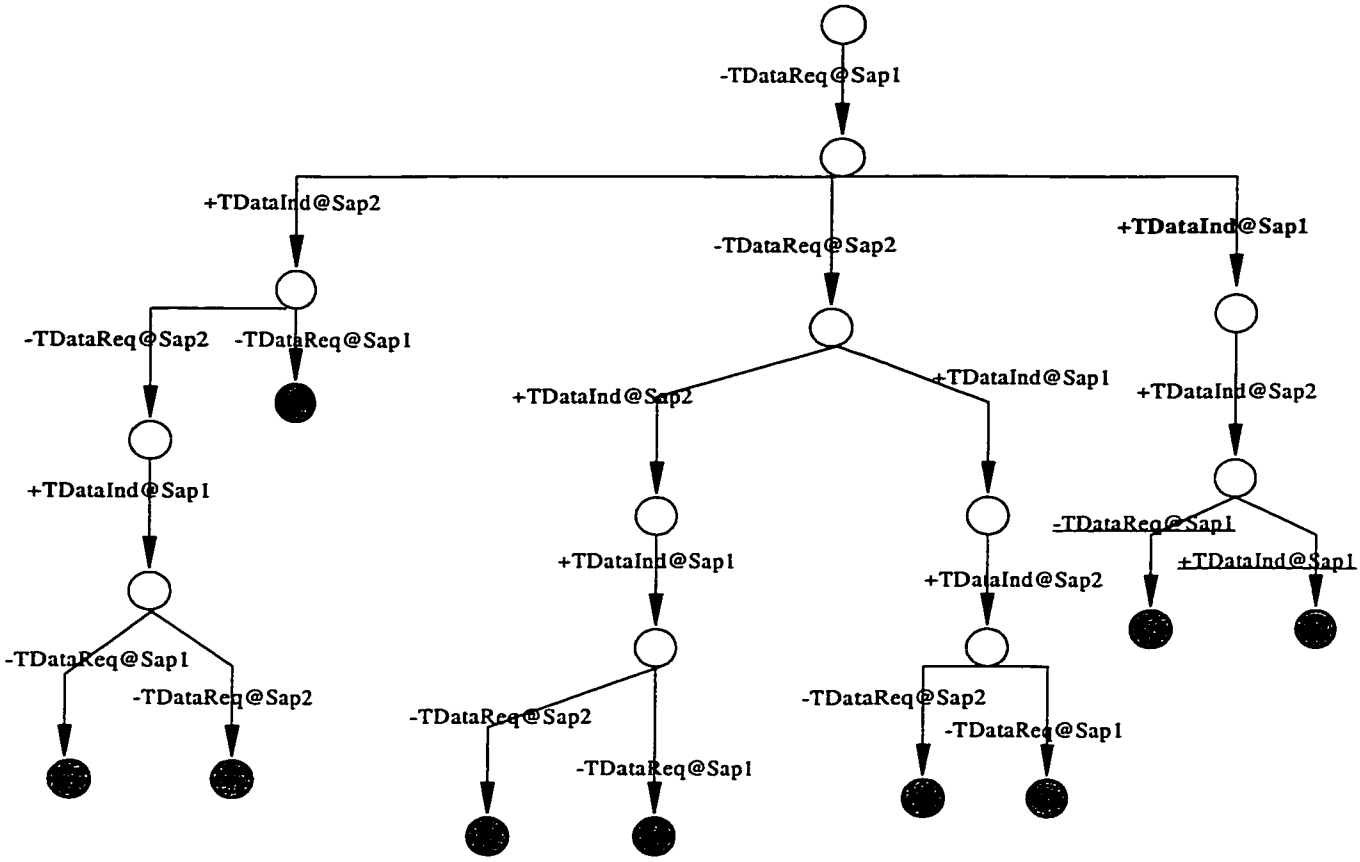


Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*



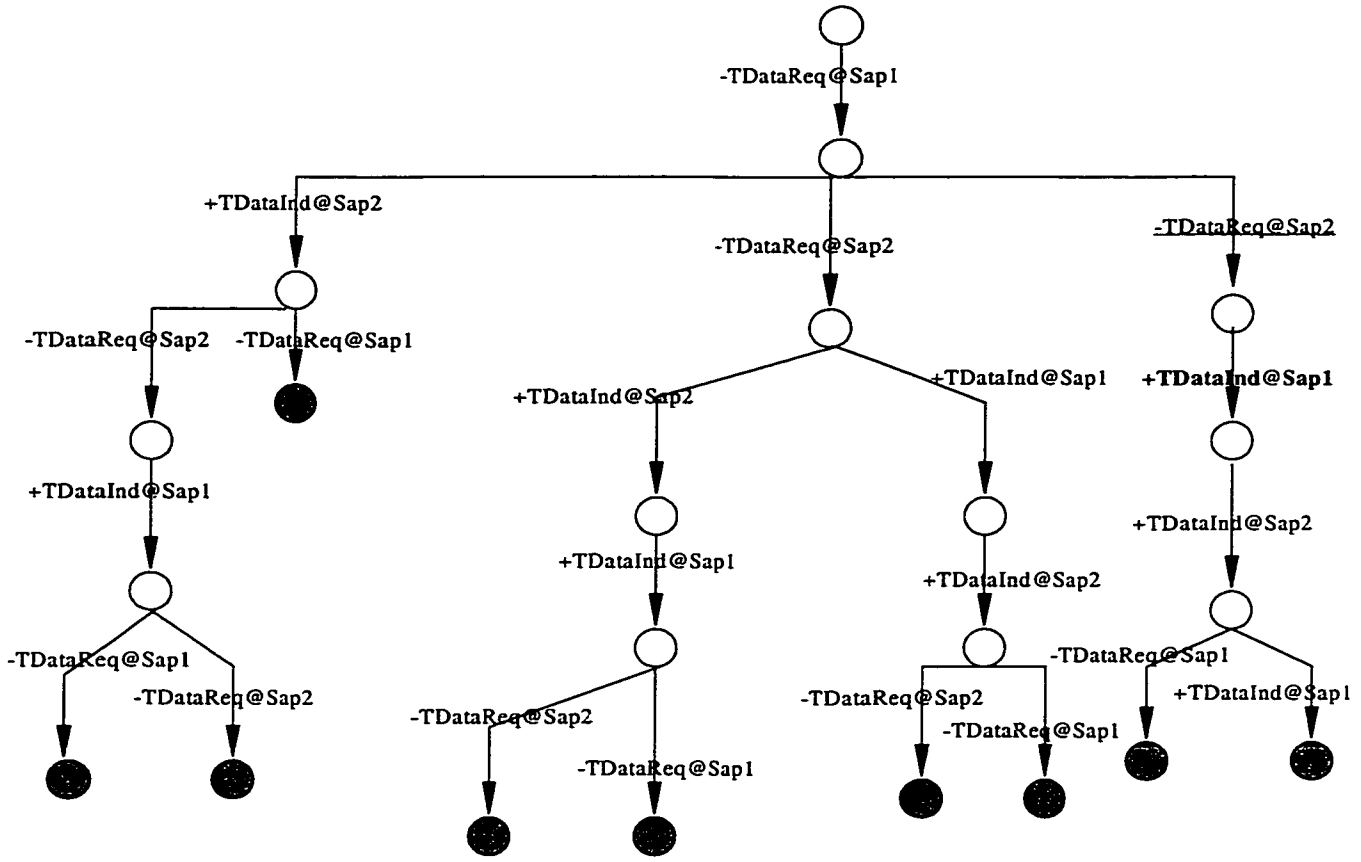
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 15:



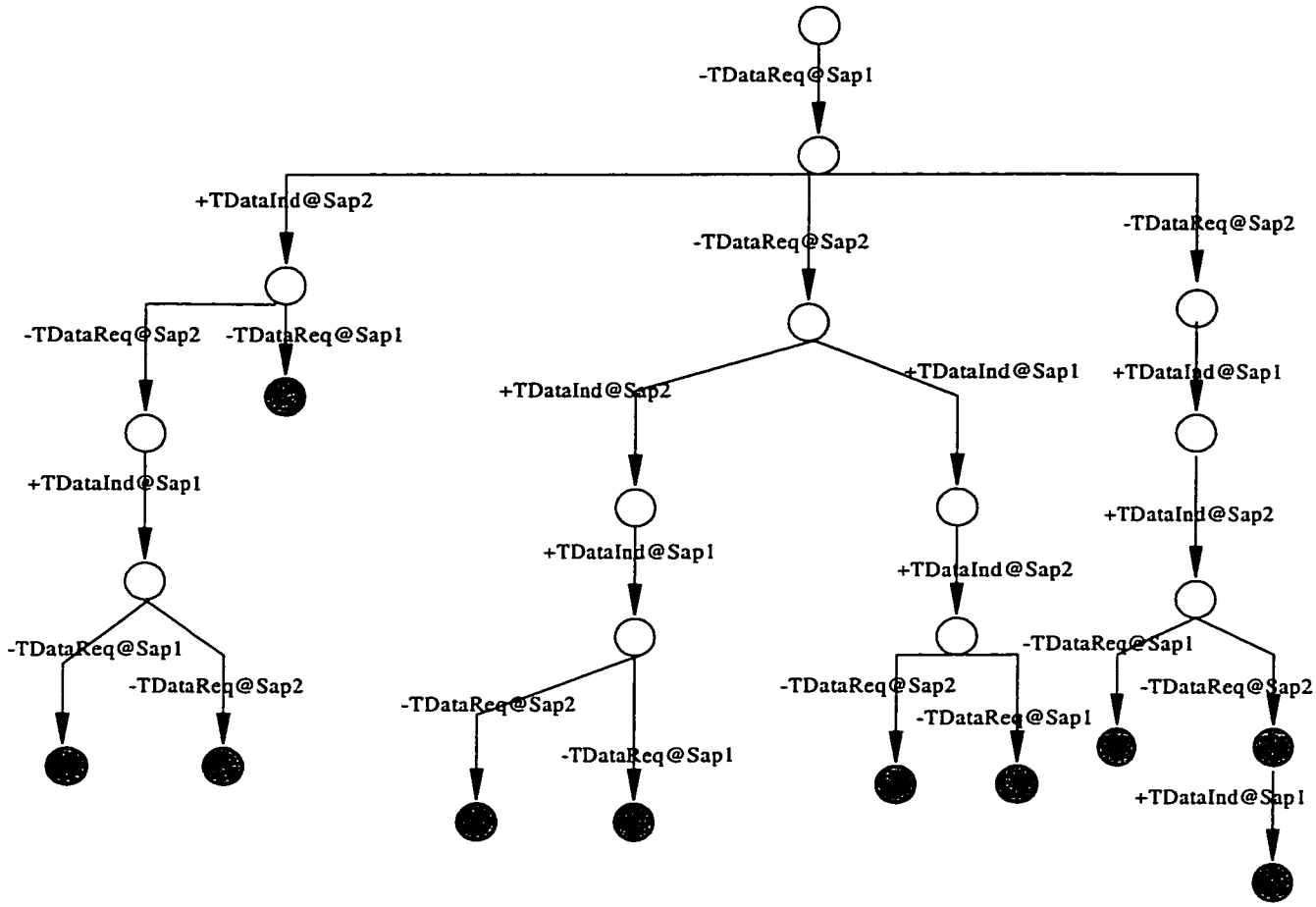
Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

pass 16:



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

Pass 17:



Note: **bold** font indicates the *current transition*
underlined font indicates the *newly added transitions*
shaded nodes are the *repeated states*

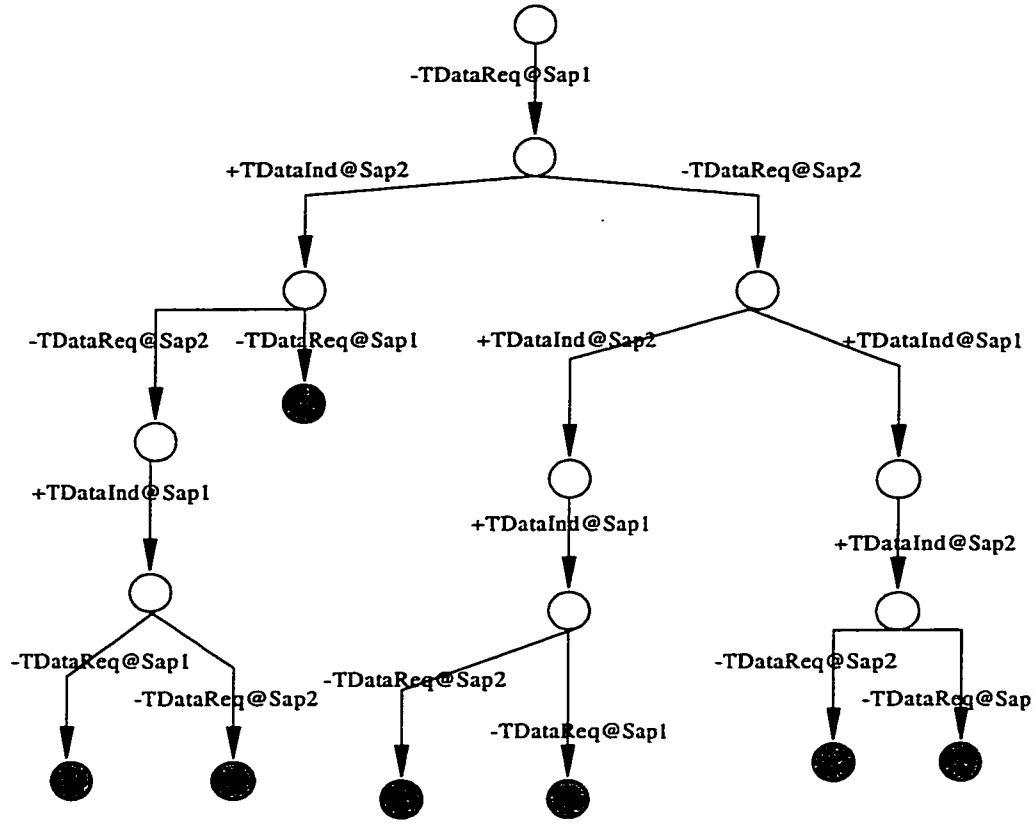


Figure 42. The Iteration Stopped (One half is generated)

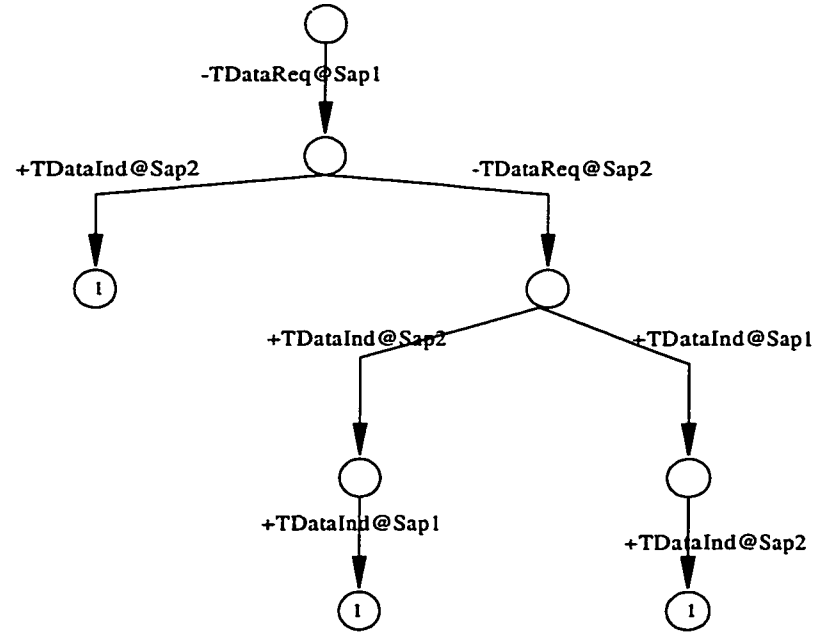


Figure 43. Labelling the initial states (One half is shown)

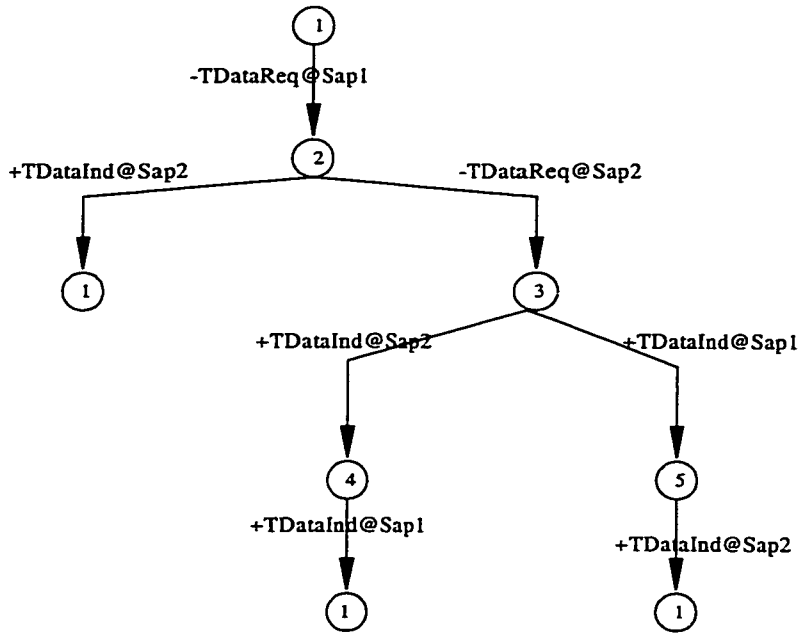


Figure 44. After Labelling all the states (Only in one half of the PSM)

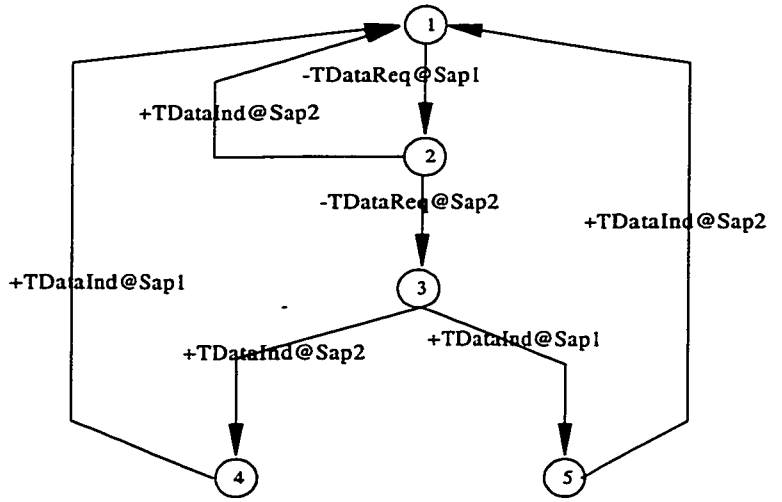


Figure 45. The PSM for Transport Connection Data Transfer Phase (Only one half of the PSM)

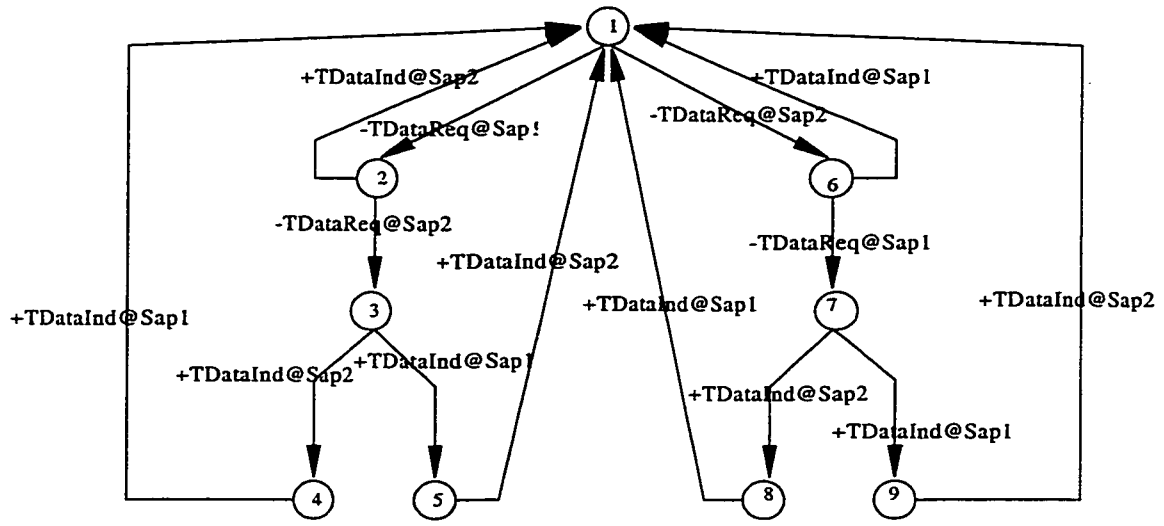


Figure 46. The complete PSM for Transport Connection Data Transfer Phase

A.3.3. Transport Connection Release Phase

The following are the inputs derived for this phase:

Σ = { -TDiscReq@SAP$_1$, -TDiscReq@SAP$_2$,
+TDiscInd@SAP$_1$, +TDiscInd@SAP$_2$ }.

I (initial events) ={ -TDiscReq@SAP$_1$, -TDiscReq@SAP$_2$,
+TDiscInd@SAP$_1$, +TDiscInd@SAP$_2$ }

LC = { }

E2EC={ (-TDiscReq@SAP$_1$, +TDiscInd@SAP$_2$),
(-TDiscReq@SAP$_2$, +TDiscInd@SAP$_1$)}

CC = { (-TDiscReq@SAP$_1$,-TDiscReq@SAP$_2$),
(-TDiscReq@SAP$_2$,-TDiscReq@SAP$_1$),
(+TDiscInd@SAP$_1$,+TDiscInd@SAP$_2$),
(+TDiscInd@SAP$_2$,+TDiscInd@SAP$_1$)}

In Figure 47, we see the construction of the PSM for this phase. Notice that a new *path* (-TDiscReq@SAP$_1$.-TDiscReq@SAP$_2$) is created when adding the event -TDiscReq@SAP$_2$, which is a member of the concurrency constraint event pair

(-TDiscReq@SAP_1, -TDiscReq@SAP_2). -TDiscReq@SAP_2 is not propagated to *path* (-TDiscReq@SAP_1.+TDiscInd@SAP_2) because there is no event pair (-TDiscReq@SAP_2,+TDiscInd@SAP_2) or (+TDiscInd@SAP_2, -TDiscReq@SAP_2) in the set of *LC*, i.e. the local properties will be corrupted if added.

In Figure 48, the PSM for the Transport Connection Release Phase is shown. Since the behavior of the phase is symmetric, only the PSM construction of the one side is shown.

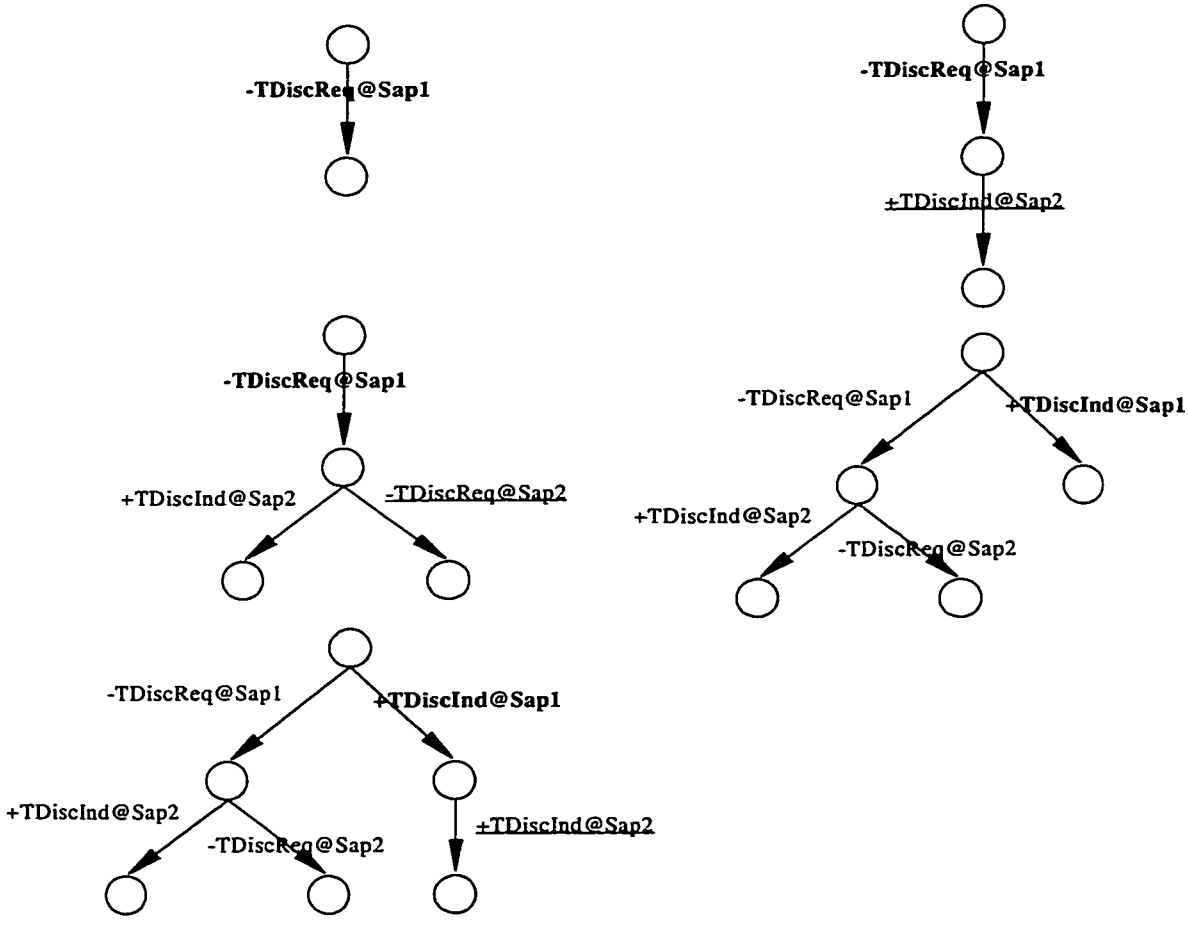


Figure 47. PSM Construction of Transport Connection Release Phase (Only one half is shown)

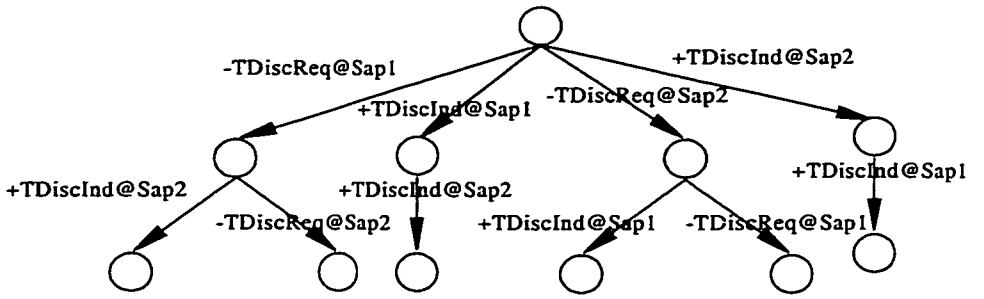


Figure 48. The PSM of the Transport Connection Release Phase

A.4. Coupling of Phases

We will assume that the interactions at the phase boundaries have been extracted as follows.

The set of event pair in the *Phase Boundary* between the Transport Connection Establishment and Transport Connection Release is:

{ (-TConReq@SAP1, -TDiscReq@SAP1),
(-TConReq@SAP2, -TDiscReq@SAP2),
(+TConInd@SAP2, -TDiscReq@SAP2),
(+TConInd@SAP1, -TDiscReq@SAP1),
(-TConRes@SAP2, -TDiscReq@SAP1),
(-TConRes@SAP1, -TDiscReq@SAP2),
(-TConRes@SAP2, -TDiscReq@SAP2),
(-TConRes@SAP1, -TDiscReq@SAP1),
(+TConConf@SAP1, -TDiscReq@SAP1),
(+TConConf@SAP2, -TDiscReq@SAP2)}

The set of event pair in the *Phase Boundary* between the Transport Connection Data Transfer and Transport Connection Release is:

{ (-TDataReq@SAP1, -TDiscReq@SAP2),
(-TDataReq@SAP1, -TDiscReq@SAP2),
(-TDataReq@SAP2, -TDiscReq@SAP1),
(-TDataReq@SAP2, -TDiscReq@SAP2),
(+TDataInd@SAP1, -TDiscReq@SAP1),
(+TDataInd@SAP2, -TDiscReq@SAP2) }

For simplicity, it will be assumed that the Transport Connection Release Phase runs until completion, i.e. all transitions to another phase is done at the final nodes.

The coupling method assumes that the communication medium is reliable and its channel contents cannot be changed. The scenario where two users can initiate a T_Disconnect Request simultaneously is excluded from the coupling example because this particular scenario implies the deletion of the channel contents. In addition, the spontaneous events (provider-initiated events) are excluded from the coupling example since a message received should have sent by the other communicating user (reliable channels).

It is also assumed that the channel capacity in either direction is limited to one message at a time.

Since the coupling rules are straight forward, how the coupling is performed and how the collision recovery transition are inserted will not be show in this section. Instead, they are discussed in Chapter 4 in detail.

In Figure 49 on page 147, the resulting GSM is depicted. It is not minimized and does not include the collision recovery transitions. They are left out to keep the

GSM simple. In Figure 50 on page 148, the minimized version of the coupled GSM is shown. Note that the collision recovery transitions are omitted again to show the difference. The GSM in Figure 51 on page 149 is the minimized GSM with the collision recovery transitions. Figures 49 - 51 depict the GSM when the channel capacity is maximum of one message in both directions.

Next, a coupled GSM where the channel capacity is not a limiting factor is shown in Figure 52 on page 150. Notice that this GSM has more transitions than the GSM in Figure 49. Finally, Figure 53 on page 151 depicts the minimized version of this GSM. Neither GSMs include the collision recovery transitions. They would be inserted in the same fashion as in Figure 51.

It is desirable to express all of the possible scenarios in an FSM which specifies a global service behavior. Since the possibility of collisions are greater in such a specification, this specification should also include the possible collision scenarios. As part of a protocol synthesis method by [Kaku 94], a set of transitions which indicate the collisions are added to the Global FSM which specifies the service. In our method, we tried to include the collision scenarios as part of the synthesis of the service specification.

The present rules for *collision recovery transitions* apply to phase boundaries. However, it is possible to expand them to include scenarios where there are collisions caused by unexpected (illegal) interactions within the same phase (e.g. issuing a T_Connect Request simultaneously at both ends of the communication channel).

Since the coupling rules are based on the channel capacity and the channel states, they have limiting factors and do not handle some exceptions. The criteria for coupling phases could be modified to consider *pending transitions* in the *phase boundaries* instead of channel contents and capacity. A *pending transition* is a transition in the present phase whose label is an event on the phase boundary and may collide with an event in the next phase if executed. Thus, removing the channel capacity limitation could result in a more complete GSM.

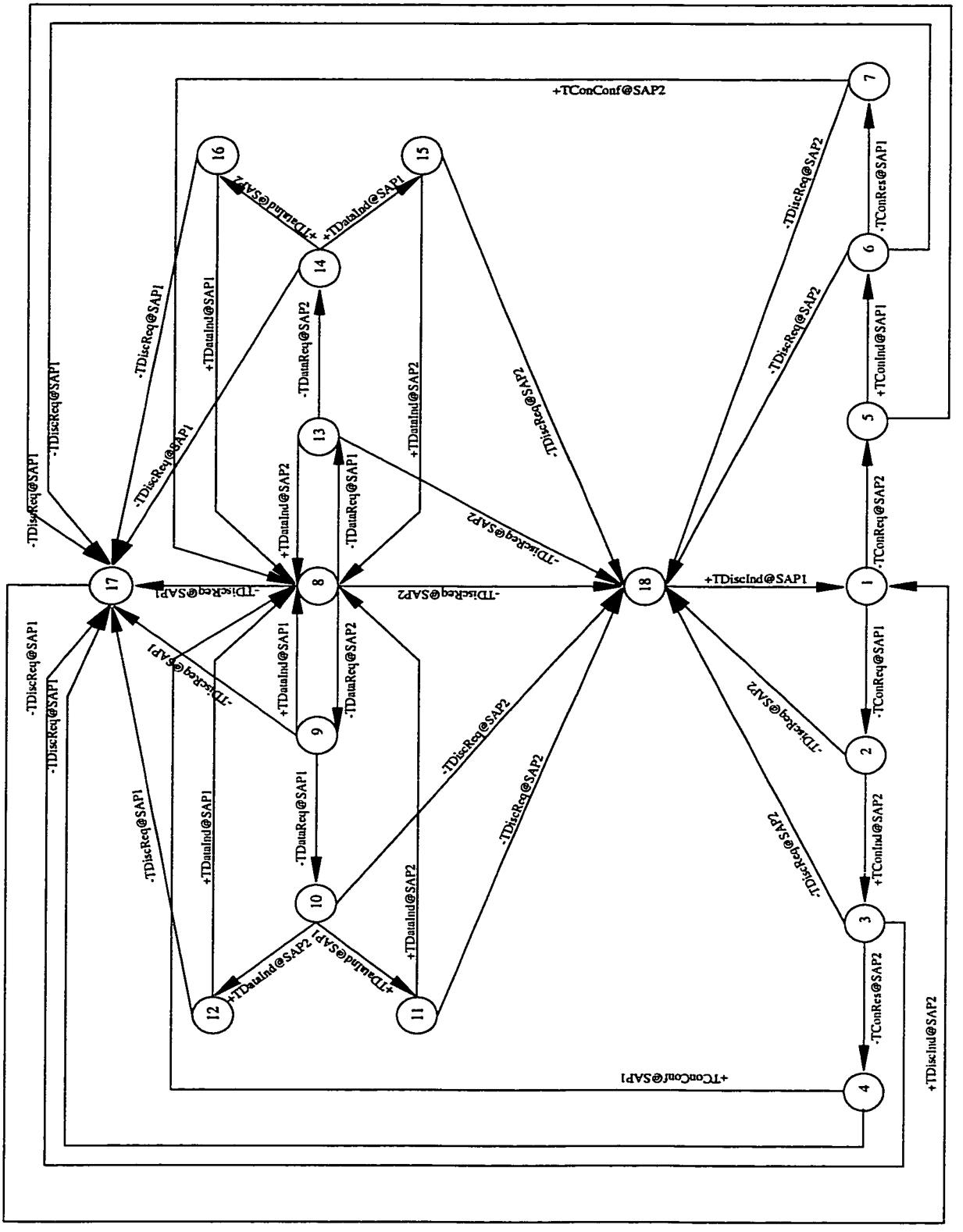


Figure 49. Coupled GSM; unminimized, channel capacity is one message, without collision recovery transitions

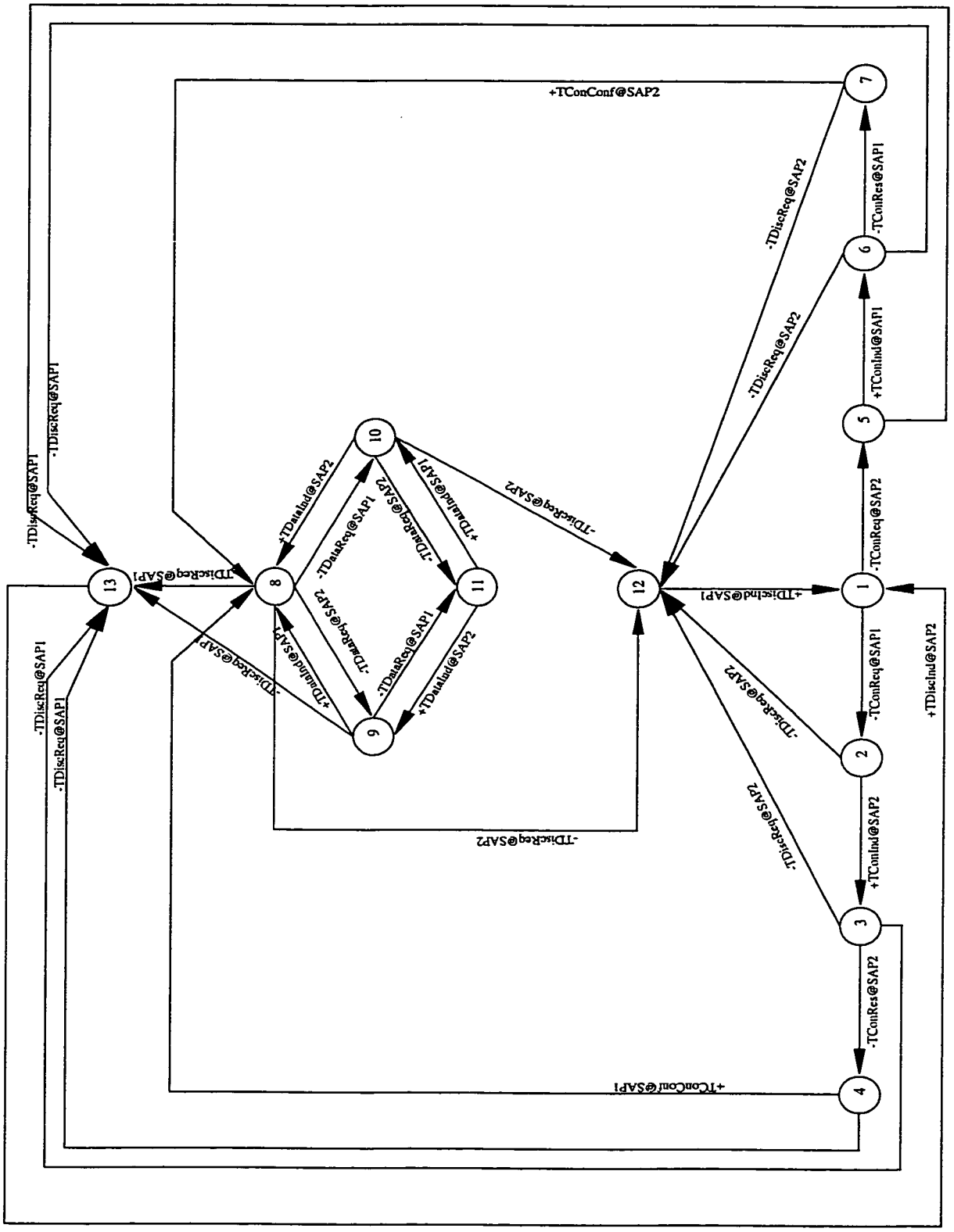


Figure 50. Coupled GSM; minimized, channel capacity is one message, without collision recovery transitions

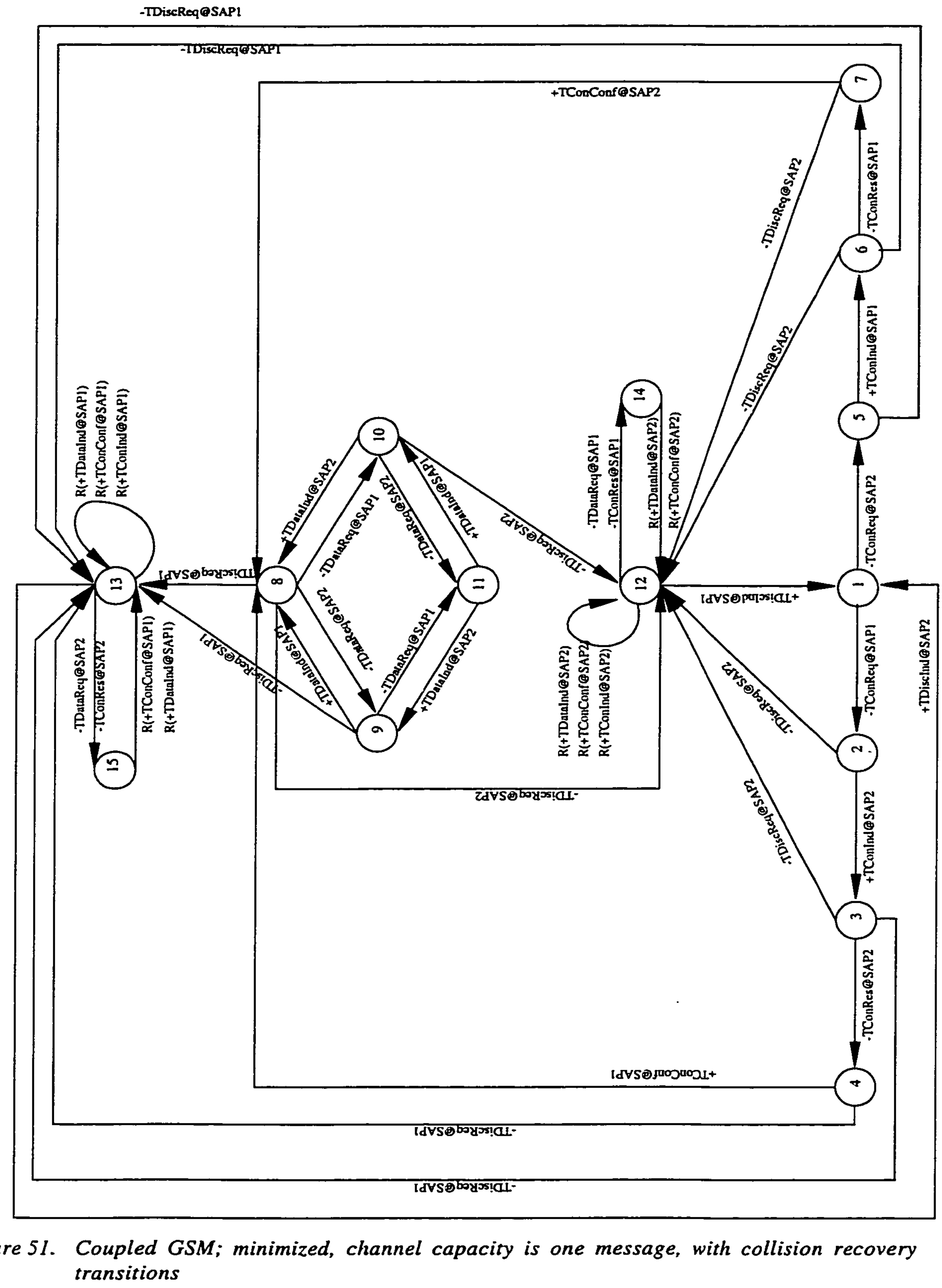


Figure 51. Coupled GSM; minimized, channel capacity is one message, with collision recovery transitions

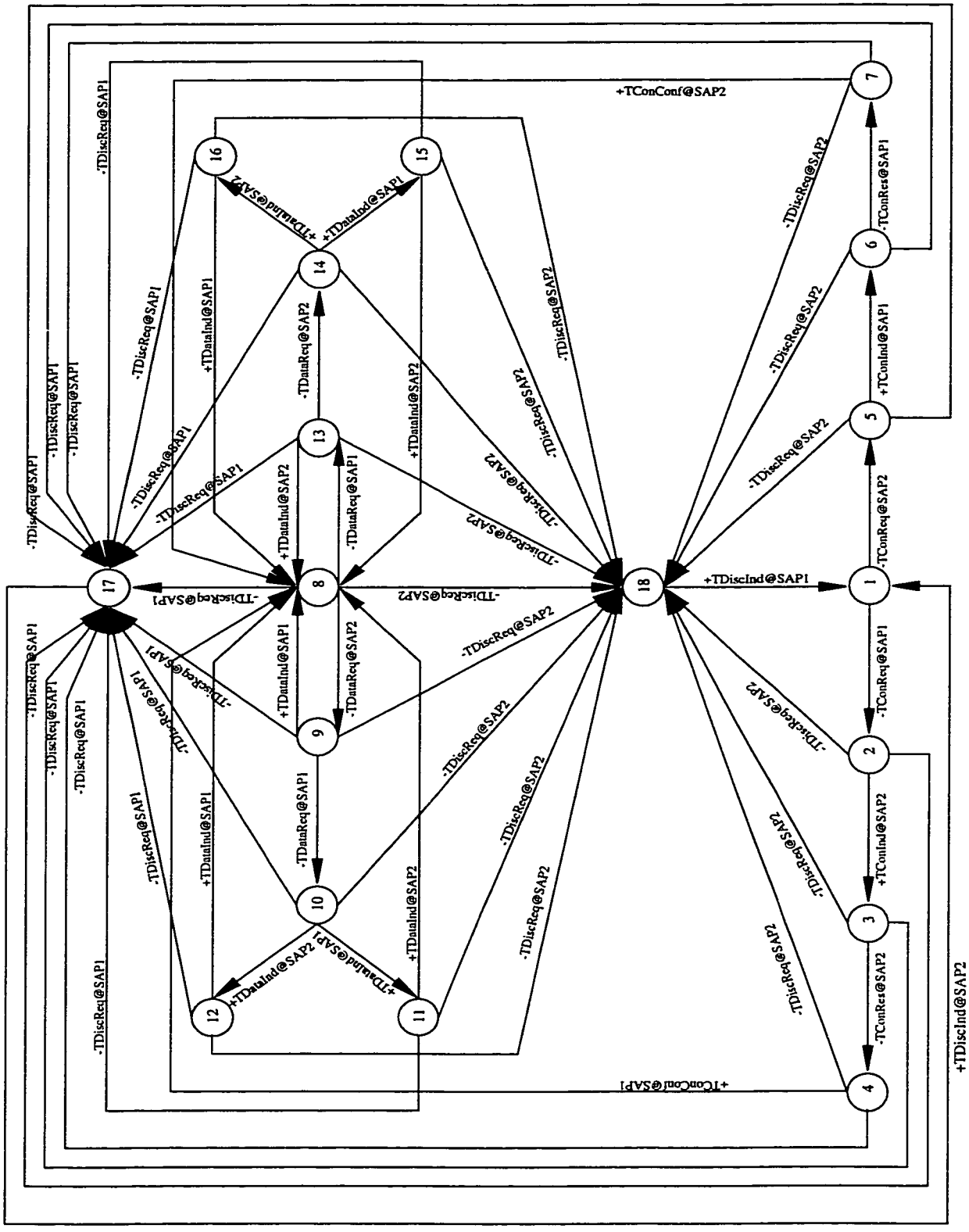


Figure 52. Coupled GSM; unminimized, channel capacity is not limited, without collision recovery transitions

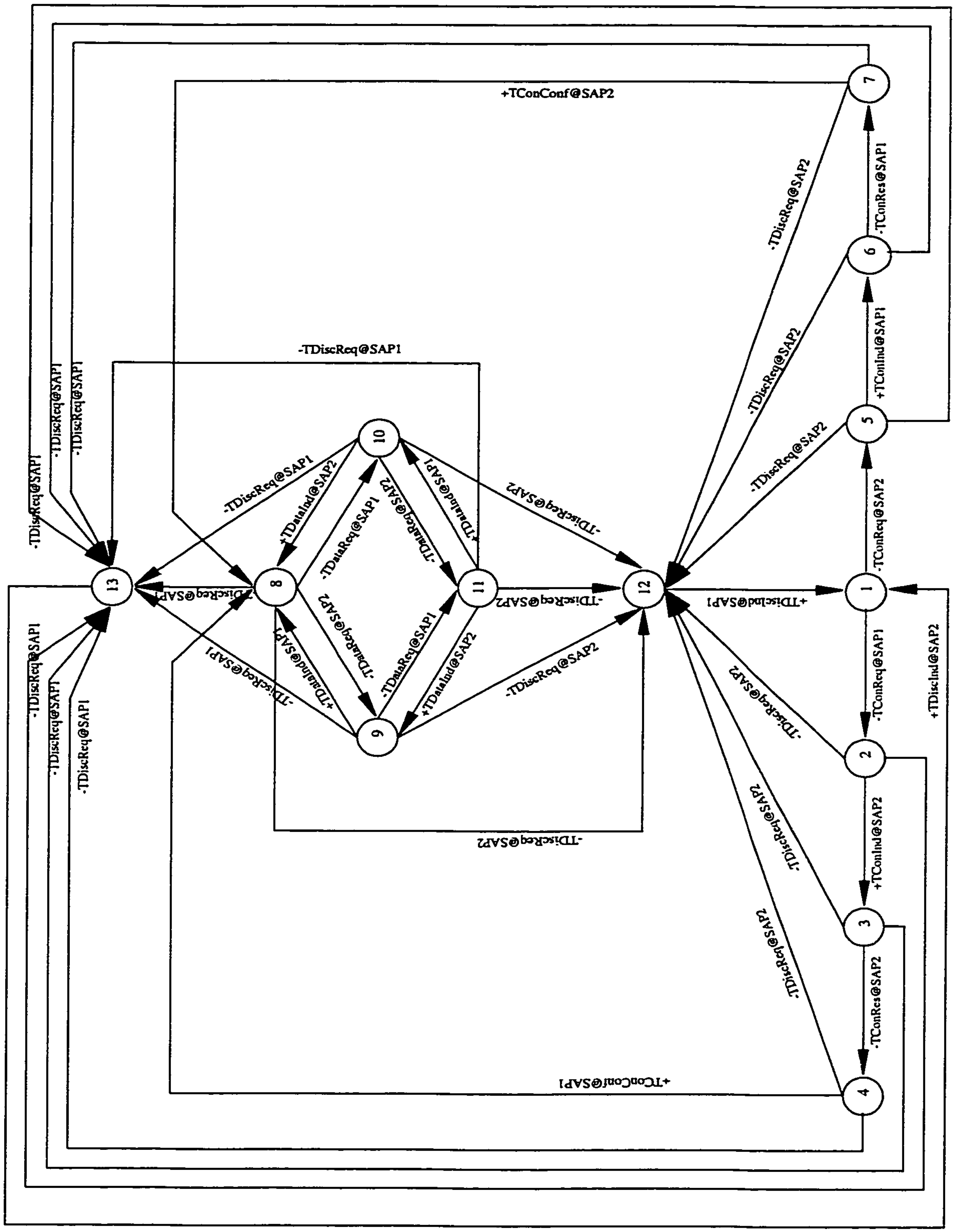


Figure 53. Coupled GSM; minimized, channel capacity is not limited, without collision recovery transitions

Appendix B. A Prototype Implementation: PSM Construction

B.1. Implementation

An earlier version of the Phase Scenario Machine Construction algorithm is given here. It constructs a phase from given set of constraints. It differs from the present algorithm, i.e. not optimized and the insertion of concurrency constraints is not handled, yet.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define MAXWORD   100
#define BUFSIZE   100

#define MAXNOPHASE      3       /* max # of phases in a service spec */
#define MAXNOEVENT      8       /* max # of events per phase */
#define MAXNOINITEVENT  4       /* max # of initial events per phase */
#define NILSTATENO      -1
#define NILEVENTINDX    -1
#define MAX_TREE_LEVEL  30

typedef enum {FALSE,
              TRUE } BOOL ;

typedef enum {  RIGHT,
                LEFT } where_to_add ;

struct eventtype {
        int     spindex;   /* index to service primitive alphabet */
        int     sapindex;  /* index to service access point alphabet */
};

struct tnode {
      int     indx2alpha ;
      int     fromstate;
      int     tostate ;
      BOOL    stopstate ;
      struct  tnode  *left;
      struct  tnode  *right;
};

char      buf[BUFSIZE] ; /*?*/
int       bufp = 0;
```

```
/* The following variable are for the current service specification we're
   working on. Initial values are assigned here for now. */

int         no_of_phases =          3 ,          /* # of phases used in the example */
            no_of_events[3] =       { 8, 4, 4},  /* # of events in each phase */
            no_of_init_events[3] = {2, 2, 4},    /* # of initial events of each
                                                                    phase */
            initevents [MAXNOPHASE][MAXNOINITEVENT]
            = { { 0, 4, NILEVENTINDX, NILEVENTINDX},
                { 0, 4, NILEVENTINDX, NILEVENTINDX },
                { 0, 1, 4, 5}
              };

char    *spalpha[] = { "nil",
                "TCONReq", "TCONInd", "TCONConf", "TCONResp",
                "TDATAReq", "TDATAInd",
                "TDISCReq", "TDISCInd"
               }; /* Service Primitive alphabet */

char    *sapalpha[] = { "nil","Sap1", "Sap2" }; /* Service Access Point Aplhabet */

struct  eventtypeeventalpha[MAXNOPHASE][MAXNOEVENT]
            = {{/* Transport Connection Establishment Phase */
                1, 1,/* TCONReq@Sap1 */
                2, 1,/* TCONInd@Sap1 */
                3, 1,/* TCONConf@Sap1 */
                4, 1,/* TCONResp@Sap1 */
                1, 2,/* TCONReq@Sap2 */
                2, 2,/* TCONInd@Sap2 */
                3, 2,/* TCONConf@Sap2 */
                4, 2 /* TCONResp@Sap2 */
                },
                {   /* Transport Connection Data Transfer Phase */
                5, 1,/* TDATAReq@Sap1 */
                6, 1,/* TDATAInd@Sap1 */
                0, 0,/* nil */
                0, 0,/* nil */
                5, 2,/* TDATAReq@Sap2 */
                6, 2,/* TDATAInd@Sap2 */
                0, 0,/* nil */
                0, 0 /* nil */
                },
                {   /* Transport Connection Release Phase */
                7, 1,/* TDISCReq@Sap1 */
                8, 1,/* TDISCInd@Sap1 */
                0, 0,/* nil */
                0, 0,/* nil */
                7, 2,/* TDISCReq@Sap2 */
                8, 2,/* TDISCInd@Sap2 */
                0, 0,/* nil */
                0, 0/* nil */
                }};


BOOL                local_constraints[MAXNOPHASE][MAXNOEVENT][MAXNOEVENT]
        = {{/* Transport Connection Establishment Phase */
                { FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE, FALSE, FALSE},
                { FALSE, FALSE, TRUE,  FALSE, FALSE, FALSE, FALSE, FALSE},
                { FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
```

```
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE}
},
{   /* Transport Connection Data Transfer Phase */
{ TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE}
},
{   /* Transport Connection Release Phase */
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE}
}},

end2end_constraints[MAXNOPHASE][MAXNOEVENT][MAXNOEVENT]
= {{ /* Transport Connection Establishment Phase */
{ FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }
},
{ /* Transport Connection Data Transfer Phase */
{ FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }
},
{ /* Transport Connection Release Phase */
{ FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }
}},
```

```
parallel_behavior_constraints[MAXNOPHASE][MAXNOEVENT][MAXNOEVENT]
= {{   /* Transport Connection Establishment Phase */
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE}
},

{   /*Transport Connection Data Transfer Phase */
{ FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ TRUE,  FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, TRUE,  FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }
},
{   /*Transport Connection Data Transfer Phase */
{ FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ TRUE,  FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, TRUE,  FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },
{ FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }
}};


/* definition of the functions */

struct tnode   *talloc(void);

void    printtree(  BOOL,
                    struct tnode *,
                    int);

void    printpath(  struct tnode    *path[],
                    int             level,
                    int             phase);

struct tnode   *addnewnode(void);

struct tnode   *copynode ( struct tnode *p);

struct tnode   *copytree(   struct tnode *);

struct tnode   *addnodebetween(  where_to_add,
                                 struct tnode *,
                                 struct tnode *);

void   delete(   struct tnode *p);
```

```
void   prunetree(   struct tnode *p);

/*
void        addbefore(struct tnode *, struct tnode *);
void        addbelow(struct tnode *, struct tnode *);
*/

struct  tnode  *init_scenario_tree ( struct tnode **,
                                     int);

void   build_scenario_tree(     struct tnode    *currentpath[],
                                struct tnode    *currentnode,
                                int             phase,
                                int             *level);


int      addlocal2tree( int             phase,
                        struct tnode    *currentnode);

void     addlocal(  struct tnode    *currentpath[],
                    struct tnode    *currentnode,
                    int             currentindx,
                    struct tnode    *subtree,
                    int             phase,
                    int             *level);

void     addtrans(      struct tnode    *path[],
                        int             lowerbound,
                        int             upperbound,
                        int             level,
                        int             indx);

BOOL  itisastopstate(  struct tnode    *currentpath[],
                        int             level,
                        int             newtrans);

void   labeltree (  struct tnode *,
                    int,
                    int *);

void propagatepath(     struct tnode    *path[],
                        int             currentindx,
                        int             level,
                        int             trans);

void addabovelevel (    struct tnode    *path[],
                        int             level,
                        int             trans);

void addbelowlevel(     struct tnode    *path[],
                        int             level,
                        int             trans);

void merge(     struct tnode *,
                int );

BOOL verifytransforward(        struct tnode    *path[],
                                int             phase,
                                int             lowlimit,
```

```
                                int             *uplimit,
                                int             trans);

BOOL verifytransbackward(       struct tnode    *path[],
                                int             phase,
                                int             *lowlimit,
                                int             uplimit,
                                int             currenttrans,
                                int             transtobeadded);

void   addend2endbelow(  struct tnode    *currentpath[],
                         struct tnode    *currentnode,
                         int             currentindx,
                         struct tnode    *subtree,
                         int             phase,
                         int             *level);

void   addend2endabove(  struct tnode    *path[],
                         struct tnode    *currentnode,
                         int             currentindx,
                         int             phase,
                         int             level);

int    addend2end2tree(  int             phase,
                         struct tnode    *currentnode,
                         BOOL            *foundbelow,
                         BOOL            *foundabove );

struct tnode  *scenariotree[MAXNOPHASE];

main ()
{
    /* struct tnode  *scenariotree[MAXNOPHASE];*/
       struct tnode*currentpath[MAX_TREE_LEVEL] ;

       int      level = 0;
       int      i, j, k;
       int      label = 0,
                initlabel = 0;

       /* for now we have initilized everything
       readinput();
       */

       printf("main\n");

       /* for every phase initilize the scnerio trees */
       for (i=0; i<no_of_phases; i++) { scenariotree[i] = NULL; }

       /* for every phase */
       for (i=0; i<no_of_phases; i++) {
           /* for every initial transition */
           for(j=0; j< no_of_init_events[i] ; j++) {
               /* init the current path*/
               level = 0;
               for (k= 0 ;k< MAX_TREE_LEVEL; k++) currentpath[k]=NULL;
               currentpath[level] = init_scenario_tree(
                                &(scenariotree[i]),
                                initevents[i][j]) ;
```

```
                printf ("scneario=%x\n", scenariotree[i]);

                build_scenario_tree(currentpath,
                        currentpath[level],
                        i,
                        &level);
                    /* printtree (TRUE, scenariotree[i]); */

            }
            /* prunetree(scenariotree[i]);
            merge ( scenariotree[i], i); */
            printf("TREE\n");
            initlabel= label++;
            labeltree (scenariotree[i], initlabel, &label);
            printtree (TRUE, scenariotree[i], i);
        }
};

struct  tnode *init_scenario_tree( struct tnode     **t,
                                   int              event0)
{

        struct tnode *temp;

        printf( "init_scenario_tree: %x , %d\n", *t, event0);

        if (*t==NULL) {
            temp = addnewnode();

            temp ->indx2alpha = event0;
            temp ->stopstate  = FALSE;
            temp ->fromstate  = NILSTATENO;
            temp ->tostate  = NILSTATENO;

            *t = temp;

            return temp;
        }
        else {
            temp = *t;
            while (temp ->right != NULL) temp = temp ->right;

            temp ->right = addnewnode();

            temp-> right ->indx2alpha = event0;
            temp-> right ->stopstate  = FALSE;
            temp-> right ->fromstate  = NILSTATENO;
            temp-> right ->tostate  = NILSTATENO;

            return temp -> right;
        }
}

void   build_scenario_tree(struct tnode *currentpath[],
                    struct tnode *currentnode,
                    int      phase,
                    int      *level)
/*      function: This procedure builds the scenario tree from a given
```

```
        set of constraints and the initial tree which includes
        the initial transitions.

    currentpath is the path from initial state to the current
            node, including the currentnode

    currentnode    is the transition whose constraints we are adding/
            verifying in the tree

    phase   is the current phase that we are building

    level    is the level of the cuurent node and number of
            nodes ( 0 to level) in the current path
*/
{
    int  indx = *level;
    BOOL addbelowcurrent = FALSE,
            addabovecurrent = FALSE ;

    int  label = 0;
    int  initlabel = 0;

    printf ("build_scenario_tree: level= %d currentnode=%x(%d)\n", *level,
                currentnode, currentnode->indx2alpha);

/*      initlabel= label++;
    labeltree (scenariotree[phase], initlabel, &label);
    printtree (TRUE, scenariotree[phase], phase);
*/
    if (currentnode != NULL) {

        if (addlocal2tree( phase, currentnode ) == TRUE) {
            /* verify the currentnode against its constraint sets */
            addlocal(       currentpath,
                            currentnode,
                            indx,
                            currentnode->left, /* subtree*/
                            phase,
                            level);
        }
        printf("-------\n");
        merge( currentnode, phase);

        if (addend2end2tree( phase,
                    currentnode,
                    &addbelowcurrent,
                    &addabovecurrent ) == TRUE) {
            if (addbelowcurrent==TRUE) {
                addend2endbelow(currentpath,
                                        currentnode,
                                        indx,
                                        currentnode->left, /* subtree*/
                                        phase,
                                        level);
                printf("--------\n");
                merge(currentnode, phase);
            }
            if (addabovecurrent==TRUE) {
                addend2endabove(currentpath,
```

```
                                        currentnode,
                                        indx,
                                        phase,
                                        *level);
                }
            }
/*
            addparallel( currentpath[],
                currentnode,
                            phase,
             level);
*/
            /*printtree(TRUE, currentpath[0], phase);*/
        if (currentnode->left!=NULL)  {
            /* we are going down in the tree */
            indx = ++*level ;

            currentpath[indx]= currentnode->left;
            build_scenario_tree(        currentpath,
                                        currentnode->left,
                                        phase,
                                        level);

            /* we are going up in the tree decrement the level */
            indx = --*level ;
        }

        if (currentnode->right!=NULL) {
            /* we are staying the same level, do not
              change the level */
            currentpath[indx]=currentnode->right;


            build_scenario_tree(        currentpath,
                                        currentnode->right,
                                        phase,
                                        level);
        }
    }
    return;
}

int     addlocal2tree(  int             phase,
                        struct tnode    *currentnode)
{

    int  i=0;
    BOOLnotfound = FALSE;

    printf( "addlocal2tree\n");
    while (( notfound =
        (local_constraints[phase][currentnode->indx2alpha][i]==FALSE))
       && (i < no_of_events[phase])) i++;
    return ( (!notfound) && (currentnode->stopstate==FALSE));
}
```

```
void    addlocal(   struct tnode    *currentpath[],
                    struct tnode    *currentnode,
                    int             currentindx,
                    struct tnode    *subtree,
                    int             phase,
                    int             *level)
/*
        function:Adds the local constraints of the current node
                 to all the paths leading from the current node

        currentpathis the path from initial state to the current
                node, including the currentnode

        subtree the rest of the below the current node

        phase   is the current phase that we are building

        level is the level of the current node and number of
        nodes ( 0 to level) in the current path
*/
{
        int  indx = *level,
             upperbound,
             lowerbound,
             i,
             j;

        int  label, initlabel;

        printf ("addlocal\n");

        /*printpath(currentpath, *level, phase);*/
        if (subtree != NULL)  {
            if (subtree -> right != NULL) {
                  addlocal(                     currentpath,
                                                currentnode,
                                                currentindx,
                                                subtree ->right,
                                                phase,
                                                level);
            }

        /*  if (eventalpha[phase][subtree->indx2alpha].sapindex !=
             eventalpha[phase][currentnode->indx2alpha].sapindex) {
        */
            if (local_constraints[phase][currentnode->indx2alpha][subtree->indx2alpha]!=
              TRUE ) {
                        indx = ++*level ;
                        currentpath[indx]= subtree;

                        if  (subtree -> stopstate!= TRUE) {

                                addlocal(       currentpath,
                                                currentnode,
                                                currentindx,
                                                subtree ->left ,
                                                phase,
                                                level);
                  }
```

```
        else /*(subtree -> stopstate!= TRUE)*/ {
            for (i=0; i<MAXNOEVENT; i++) {
                if (local_constraints[phase]
                        [currentnode -> indx2alpha]
                        [i]) {
                    /* add the constraint to the current path */
                    /* add from current node to present */
                    lowerbound = currentindx ;
                    upperbound = *level;
                    if ( verifytransforward
                            (    currentpath,
                                 phase,
                                 currentindx,
                                 &upperbound,
                                 i) ) {

                        printpath(currentpath, *level, phase);
                        addtrans(       currentpath,
                                        lowerbound,
                                        upperbound,
                                        *level,
                                        i);
                        initlabel=0;
                        label = 0;
                        initlabel= label++;
                        labeltree (scenariotree[phase], initlabel, &label);
                        printtree (TRUE, scenariotree[phase], phase);
                    }

                }
            }
        }
        indx = --*level ;
    }
}
else { /* (subtree != NULL) */
    printf("addlocal: phase= %d indx2alpha= %d \n",
                phase, currentnode -> indx2alpha);

    for (i=0; i<MAXNOEVENT; i++) {
        if (local_constraints[phase]
                        [currentnode -> indx2alpha]
                        [i]) {
            /* add the constraint to the current path */
            /* add from current node to present and below*/
            lowerbound = currentindx ;
            upperbound = *level;
            if ( verifytransforward
                    (               currentpath,
                                    phase,
                                    currentindx,
                                    &upperbound,
                                    i) ) {

                printpath(currentpath, *level, phase);
                addtrans(       currentpath,
                                lowerbound,
                                upperbound,
                                *level,
```

```
                                        i);
                        initlabel=0;
                        label = 0;
                        initlabel= label++;
                        labeltree (scenariotree[phase], initlabel, &label);
                        printtree (TRUE, scenariotree[phase], phase);
                    }
                }
            }
        }
}

int   addend2end2tree(   int            phase,
                         struct tnode    *currentnode,
                         BOOL            *foundbelow,
                         BOOL            *foundabove      )
{

    int   i=0;

    *foundbelow = FALSE;
    *foundabove = FALSE;

    printf( "addend2end2tree\n");
    while ( i < MAXNOEVENT) {

      if (end2end_constraints[phase][currentnode->indx2alpha][i] == TRUE) {
                *foundbelow = TRUE;
      }
      if (end2end_constraints[phase][i][currentnode->indx2alpha] == TRUE) {
                *foundabove = TRUE;
      }
      i++ ;
    }

    return (( *foundbelow && (currentnode->stopstate==FALSE)) ||
          *foundabove);
}

void  addend2endbelow( struct tnode    *currentpath[],
                       struct tnode    *currentnode,
                       int             currentindx,
                       struct tnode    *subtree,
                       int             phase,
                       int             *level)
/*
    function:     Adds the local constraints of the current node
              to all the paths leading from the current node

    currentpath   is the path from initial state to the current
              node, including the currentnode

    subtree       the rest of the below the current node

    phase         is the current phase that we are building

    level         is the level of the current node and number of
              nodes ( 0 to level) in the current path
*/
```

```
{
    int    indx = *level,
           i,
           j;

    int    lowerbound,
           upperbound = 0;
    int    label, initlabel;
          /*printpath(currentpath, *level, phase);*/

    if (subtree != NULL)  {

              printf("addend2endbelow level=%d current= %d subtree= %d subtreestop= %d\n",
                        *level, currentnode->indx2alpha, subtree->indx2alpha,
                        subtree -> stopstate);

              if (subtree -> right != NULL) {
              /* currentpath[indx]=subtree;*/

                   addend2endbelow(          currentpath,
                                             currentnode,
                                             currentindx,
                                             subtree ->right,
                                             phase,
                                             level);
              }

              if (end2end_constraints
                        [phase][currentnode->indx2alpha][subtree->indx2alpha] !=
                        TRUE ) {
                   indx = ++*level ;
                   currentpath[indx]= subtree;

                   if  (subtree -> stopstate!= TRUE) {
                        printf ("level= %d\n", *level);
                        addend2endbelow(          currentpath,
                                                  currentnode,
                                                  currentindx,
                                                  subtree ->left ,
                                                  phase,
                                                  level);
                   }
                   else /*(subtree -> stopstate!= TRUE)*/ {
                        for (i=0; i<MAXNOEVENT; i++) {
                             if (end2end_constraints[phase]
                                                  [currentnode -> indx2alpha]
                                                  [i]) {
                                  /* add the constraint to
                                       the current path */
                                  /* add from current node
                                       to present */
                                  lowerbound = currentindx;
                                  upperbound = *level;
                                  if (verifytransforward
                                            (    currentpath,
                                                 phase,
                                                 currentindx,
                                                 &upperbound,
                                                 i)) {
```

```
                                printpath(currentpath, *level, phase);
                                addtrans(       currentpath,
                                                lowerbound,
                                                upperbound,
                                                *level,
                                                i);
                                initlabel=0;
                                label = 0;
                                initlabel= label++;
                                labeltree (scenariotree[phase], initlabel, &label);
                                printtree (TRUE, scenariotree[phase], phase);
                            }
                        }
                    }
                }
                indx = --*level ;
            }
        }
        else { /* (subtree != NULL) */
            printf("addend2end: phase= %d indx2alpha= %d level= %d \n",
            phase, currentnode -> indx2alpha, *level);

            for (i=0; i<MAXNOEVENT; i++) {
                if (end2end_constraints[phase]
                    [currentnode -> indx2alpha]
                    [i]) {
                /* add the constraint to the current path */
                /* add from current node to present and below*/
                lowerbound = currentindx;
                upperbound = *level;
                if ( verifytransforward
                            (   currentpath,
                                phase,
                                currentindx,
                                &upperbound,
                                i) ) {

                    printpath(currentpath, *level, phase);
                    addtrans(       currentpath,
                                        lowerbound,
                                        upperbound,
                                        *level,
                                        i);
                    initlabel=0;
                    label = 0;
                    initlabel= label++;
                    labeltree (scenariotree[phase], initlabel, &label);
                    printtree (TRUE, scenariotree[phase], phase);
                }
                }
            }
        }
    }
```

```
void   addend2endabove(      struct tnode   *path[],
                     struct tnode   *currentnode,
                     int            currentindx,
                     int            phase,
                     int            level)
/*
     function:      Adds the local constraints of the current node
                to all the paths leading from the current node

     currentpath    is the path from initial state to the current
                node, including the currentnode

     subtree        the rest of the below the current node

     phase          is the current phase that we are building

     level          is the level of the current node and number of
                nodes ( 0 to level) in the current path
*/

{
     int    i=0,
          j=0;

     int lowerbound = 0,
        upperbound;
     int label, initlabel;

        printf("addend2endabove: phase= %d indx2alpha= %d \n",
                     phase, currentnode -> indx2alpha);

        if (currentindx == 0) {
            while ( j< MAXNOINITEVENT) {
                if (path[currentindx]->indx2alpha == initevents[phase][j])  {
                    break;
                }
                j++;
            }
            if (j == MAXNOINITEVENT) {
                printf ("ERROR: initialization corrupted\n");
            }
        }

        for (i=0; i<MAXNOEVENT; i++) {
            if (end2end_constraints                 [phase]
                                                    [i]
                                                    [currentnode -> indx2alpha] ) {
                /* add the constraint to the current path */
                /* add from current node to present and below*/
                lowerbound = 0 ;
                upperbound = currentindx-1 ;
                printf ("i=%d current=%d\n", i, currentnode -> indx2alpha);
                if ( verifytransbackward
                            (   path,
                                phase,
                                &lowerbound,
                                upperbound,
                                currentnode -> indx2alpha,
                                i) ) {
```

```
                printpath(path, level, phase);
                addtrans(           path,
                                    lowerbound,
                                    upperbound,
                                    level,
                                    i);
                initlabel=0;
                label = 0;
                initlabel= label++;
                labeltree (scenariotree[phase], initlabel, &label);
                printtree (TRUE, scenariotree[phase], phase);
            }
        }
    }
}

BOOL verifytransforward( struct tnode   *path[],
                int     phase,
                int         lowlimit,
                int         *uplimit,
                int         trans)
{
    int  i = lowlimit;
    BOOLverified = FALSE;

    printf ("verifytransforward: lowerlimit=%d upperlimit=%d \n", lowlimit,
                *uplimit);

    while (i<= *uplimit) {
        if (((local_constraints[phase][path[i]->indx2alpha][trans] == FALSE) &&
             (end2end_constraints[phase][path[i]->indx2alpha][trans] == FALSE) &&
         (parallel_behavior_constraints[phase][path[i]->indx2alpha][trans]
            == FALSE)) || (path[i]->stopstate == TRUE)) {
            break;
        }
        i++ ;
    }
    i--;
    if (( i <= *uplimit) && ( i>=lowlimit)) {
        *uplimit = i;
        return TRUE;
    }
    else {
        printf ("verifytransforward: %d is the upper limit\n", i);
        return FALSE;
    }
}

BOOL verifytransbackward(           struct tnode   *path[],
                                    int         phase,
                                    int         *lowlimit,
                                    int         uplimit,
                                    int         currenttrans,
                                    int         transtobeadded)
{
    int    i = uplimit;
    BOOL found=FALSE;

    if (i<0) {
```

```
            printf ("ERROR: verification boundaries out of range i=%d\n", i);
            return FALSE;
        }

        while ( i >= *lowlimit) {

            if (path[i]->indx2alpha == transtobeadded) {
                found = TRUE ;
                break;
            }

            if (path[i]->indx2alpha == currenttrans) break;

            if ( (local_constraints[phase][path[i]->indx2alpha][transtobeadded] == FALSE) &&
                (end2end_constraints[phase][path[i]->indx2alpha][transtobeadded] == FALSE)
                &&
                (parallel_behavior_constraints[phase][path[i]->indx2alpha][transtobeadded]
                    == FALSE)) {
                printf("-------\n");

                break;
            }

            i-- ;
        }
        i++ ;
        if ((!found) && ( i >= *lowlimit)  && ( i <= uplimit)) {
            *lowlimit= i ;
            return TRUE;
        }
        else {
            if (found) printf ("verifytransbackward: FOUND\n");
            else {
                printpath(path, uplimit, phase);
                printf ("verifytransbackward: lowerlimit=%d upperlimit=%d i=%di trans=%d\n",
                        *lowlimit, uplimit, i, transtobeadded);
            }
            return FALSE;
        }
}

void propagatepath(struct tnode   *path[],
                   int      currentindx,
                   int      level,
                   int      trans)
{
        struct tnode*new,
                    *temp,
                    *clone;

        BOOL   stop=itisastopstate( path, level, trans);

        int i, j, upperlimit;

        printf ( "propagatepath: current= %d level= %d trans= %d \n",
                        currentindx, level, trans);

        if ( path[level] -> stopstate != TRUE ) {
            upperlimit = level ;
```

```
        }
        else {
            upperlimit = level -1 ;
        }


        /* add the new node after path[level]
        */
        for (i=currentindx+1; i<=upperlimit; i++)
        {
            /* go to the most right of the current node
            */
            temp = path[i] ;
            while ( temp -> right != NULL) temp = temp -> right ;

            /* add a new node to the most right of
            the current node
            */
            new=addnodebetween(RIGHT, temp, temp->right);
            new -> indx2alpha = trans;
            new -> stopstate  = stop;
            new -> fromstate  = NILSTATENO;
            new -> tostate  = NILSTATENO;

            /* copy the remaining of the current path
            to the left of the new node
            */
            clone = new ;
            for(j=i ; j<=level ; j++)
            {
                clone -> left = copynode ( path[j] );
                clone = clone -> left ;
            }
            if (path[level] -> left != NULL)
            clone -> left = copytree( path[j] -> left);
        }
}

void addabovelevel ( struct tnode   *path[],
                                int     level,
                                int     trans)
{
        struct tnode  *new,
                      *temp,
                      *prev;

        printf("addabovelevel:%d\n", trans);
        prev = path[level-1];
        temp = path[level-1] -> left;

        while ( (temp != NULL) && ( temp != path[level]) ) {
            prev = temp;
            temp = temp -> right ;
        }

        if (temp == path[level]) {
            /* this is the first time we are adding
              a transition before level.
            */
```

```
            if ( temp == path[level-1]-> left) {
                /* add to left of level-1 and before level */

                new=addnodebetween (LEFT, prev, prev -> left);
            }
            else { /* add to the right of prev */
                new=addnodebetween(RIGHT, prev, prev->right);
            }
        }
        else { /* We have already added at least one transition before level */
            new=addnodebetween(RIGHT, prev, prev->right);
            new -> left = copytree(path[level]);
        }

        new -> indx2alpha = trans;
        new -> stopstate  = itisastopstate( path, level, trans);
        new -> fromstate  = NILSTATENO;
        new -> tostate    = NILSTATENO;
}



void addbelowlevel(   struct tnode   *path[],
                int          level,
                int          trans)
{
        struct tnode      *new,
                           *temp,
                          *prev;

        printf ("addbelowlevel: %d\n", trans);

        prev = path[level];
        temp = path[level] -> left ;

        while (temp != NULL) {
            prev = temp ;
            temp = temp -> right;
        }

        if (prev == path[level]) {
            /* add to the left of level */
            new = addnodebetween( LEFT, prev, temp);
        }
        else {
            /* add to the most right of level->left */
            new = addnodebetween( RIGHT, prev, temp);
        }
        new -> indx2alpha = trans;
        new -> stopstate  = itisastopstate( path, level, trans);
        new -> fromstate  = NILSTATENO;
        new -> tostate    = NILSTATENO;

}
```

```
void addtrans(struct tnode *path[],
                    int     lowerbound,
                    int     upperbound,
                    int     level,
                    int     trans)

/*    function: This procedure takes a path made up of "level"
            number of elements and adds a new node in front
            of every element of the path in the tree and copies
            the remaning part of the path and the following subtree
            to the created new branch.
      path   the current path we are evaluating
      level  is the depth level of the current node, also the number
           of elements in the path
*/
{
        int  i, j, upperlimit;
        struct tnode*new,
                    *temp,
                    *clone;

        BOOL  stop = FALSE;

        printf( "addtrans: lowerbound=%d upperbound=%d level=%d\n",
                    lowerbound, upperbound, level);

        if ( path[level]->stopstate == TRUE ) {
            printf("addtrans:STOP\n");

            if ( (upperbound >= level) ||
                (lowerbound > upperbound) ||
                ((itisastopstate( path, level, trans) == TRUE) &&
                ((upperbound+1)<level))) {
                printf( "ERROR: lowerbound=%d upperbound=%d level=%d\n",
                                lowerbound, upperbound, level);
            }
            else /* upperbound < level */
            {
                if ( lowerbound==upperbound ) {
                    /* below upperbound */
                    addabovelevel( path, (upperbound+1), trans);
                }
                else /* lowerbound<upperbound */ {
                    if ( itisastopstate( path, level, trans)
                       != TRUE ) {
                        /* propagate */
                        propagatepath(          path,
                                                lowerbound,
                                                upperbound,
                                                trans);
                    }
                    /* add above level */
                    addabovelevel( path, (upperbound+1), trans);

                }
            }
        }
        else {
                printf( "addtrans:NOTSTOP\n");
```

```
                if (        (upperbound > level) ||
                            (lowerbound > upperbound) ||
                            ((itisastopstate( path, level, trans) == TRUE) &&
                            (upperbound != level))) {
                    printf( "ERROR: lowerbound=%d upperbound=%d level=%d\n",
                                lowerbound, upperbound, level);
                }
                else if ( itisastopstate( path, level, trans) != TRUE ) {
                    if ( lowerbound == upperbound ) {
                        if (upperbound<level) {
                            /* add below upperbound*/
                            addabovelevel( path,
                                        (upperbound+1),
                                        trans);
                        }
                        else { /* upperbound == level */
                            addbelowlevel( path, level, trans);
                        }
                    }
                    else if ( lowerbound < upperbound) {
                        if ( upperbound < level ) {
                            propagatepath(      path,
                                                lowerbound,
                                                upperbound,
                                                trans);
                            addabovelevel(      path,
                                                (upperbound+1),
                                                trans);
                        }
                        else /* upperbound == level */ {
                            propagatepath(      path,
                                                lowerbound,
                                                upperbound,
                                                trans);
                            addbelowlevel( path, level, trans);
                        }
                    }
                }
                else {
                    if ( upperbound == level) {
                        /* add below level */
                        addbelowlevel( path, level, trans);
                    }
                }
        }
        return ;
}


void    merge(struct tnode *current, int phase)
{
        struct tnode *temp,
                *prev,
                *obsolete,
                *new;

        temp = current -> right;
```

```
        prev = current;

        /*printf( " current (%x) = %d ", current, current ->indx2alpha);*/
        while (temp != NULL) {
            /* printf( " temp(%x) = %d ", temp, temp ->indx2alpha);*/
            if ( current ->indx2alpha != temp ->indx2alpha ) {
                prev = temp;
                temp = temp->right;
            }
            else { /*printf (" break\n"); */ break;}
        }

        if (temp != NULL) {
            /* we have found a redundant node */
            obsolete = temp ;
            prev->right = obsolete-> right;

            if (( current->left != NULL) && ( obsolete->left != NULL)) {

                temp = current->left;
                while (temp -> right != NULL) temp = temp -> right;
                temp -> right = obsolete -> left;

            }
            else if ((current->left != NULL) && (obsolete->left == NULL)) {

                temp = current->left;
                 while (temp -> right != NULL) temp = temp -> right;

                 new=addnodebetween(RIGHT, temp, temp->right);
            }
            else if ((current->left == NULL) && (obsolete->left != NULL)) {

                new=addnodebetween(LEFT, current, current->left);

                new -> right = obsolete -> left ;

            }
            else if ((current->left == NULL) && (obsolete->left == NULL)) {
                /* no need to add nil */
            }
            free(obsolete);
        }
        /*printtree (TRUE, current, phase);*/
        if (current -> right != NULL ) merge(current -> right, phase);
        if (current -> left  != NULL ) merge(current -> left, phase);

        return ;
}



BOOL itisastopstate( struct tnode *currentpath[], int level, int newtrans)
{
        int  i=0;
        BOOLstop = FALSE;

        printf("itisastopstate\n");
```

```
        while (i<=level) {
            if (newtrans == currentpath[i]->indx2alpha) {
                stop = TRUE;
                break ;
            }
            i++;
        }
        printf ("i=%d level=%d newtrans=%d stop=%d \n",
                        i, level, newtrans, stop);
        return (stop) ;
}

void    printpath ( struct tnode    *path[],
                    int             level,
                    int             phase)
{
        int i;

        printf("phase=%d \n", phase);

        for (i=0; i<=level; i++)
        {

            printf( " i:%d path[i]=%x indx2alpha=%d stopstate=%d\n",
                            i, path[i], path[i]->indx2alpha, path[i]->stopstate);
        }
}

struct  tnode  *addnodebetween(  where_to_add  location,
                                 struct tnode     *father,
                                 struct tnode     *child )
/*     function: This procedure adds a new node between
            the father and the child.
*/
{
     struct tnode *new;

            printf("addnodebetween\n");
            new = addnewnode();

             new   -> left = child;

            if (location == RIGHT) {
              father -> right = new;
              }
            else {
              father -> left = new;
            }

            if (child != NULL) {
                new   -> right = child -> right;
                child -> right = NULL;

            }
            return new;

}
```

```
struct  tnode   *copytree( struct tnode *p)
{
      struct  tnode   *new;

         printf("copytree\n");

      new = copynode(p);
      if (p->left != NULL) copytree(p->left);
      if (p->right != NULL) copytree(p->right);
      return new;
}

struct  tnode   *copynode ( struct tnode *p)
{
      struct tnode *new;

      printf("copynode\n");

      new = addnewnode();

      new-> indx2alpha= p-> indx2alpha;
      new-> stopstate= p-> stopstate;

      return new;
}

struct  tnode   *addnewnode(void)
{
         struct  tnode   *new;


         new= talloc();

         printf("addnewnode = %x\n", new);

         new->left      = NULL;
         new->right     = NULL;

         new->fromstate  = NILSTATENO;
         new->tostate    = NILSTATENO;
         new->stopstate  = FALSE;
         new->indx2alpha = NILEVENTINDX;
         return new;
}

struct tnode *talloc(void)
{
     return (struct tnode *) malloc(sizeof(struct tnode));
}

void   labeltree (struct tnode *p, int from, int *to)
{
         if ( p!= NULL) {
              p->fromstate = from;
              p->tostate = *to;
              *to= *to + 1;

              if ( p->left != NULL ) {
```

```
                labeltree (p->left, *to - 1 , to);
            }
            if ( p->right != NULL ) {
                labeltree (p->right, from, to);
            }
        }
        return;
}

void    delete(struct tnode *p)
{
        if ( p -> left !=  NULL) {
            delete (p -> left);
        }
        if ( p -> right != NULL) {
            delete (p->right);
        }
        free(p);
}

void    prunetree( struct tnode *p)
{
        if (p != NULL) {
            if ( p -> stopstate == TRUE ) {
                if ( p -> left != NULL) delete( p->left) ;
                p->left = NULL;
            }
            prunetree( p-> left);
            prunetree( p-> right);
        }
}

void  printtree(    BOOL          first,
                    struct tnode  *p,
                    int           phase)
{
        struct tnode *temp;

        int     count;

        if (p != NULL) {
          if ( first ) {
            /* print the header */
                printf (
                        "\nFrom State  : ( Transition, To State) Pairs\n");
                printf (
                        "------------  --------------------------------\n");
                printf ("    %2d    : ", p -> fromstate);

                temp = p;
                count=0 ;
                while ( temp != NULL)  {
                    if ( count++ >= 2) {
                        printf ("\n          : ");
                        count = 0;
                    }
                    printf ("( %x, %-9s %1s %-5s , %2d )  ",
                        temp,
                    /* printf ("( %-9s %1s %-5s , %2d )  ", */
```

```
                    spalpha[eventalpha[phase][temp->indx2alpha].spindex],
                        "@",
                    sapalpha[eventalpha[phase][temp->indx2alpha].sapindex],
                    temp -> tostate);
                temp = temp -> right;
            }
            printf ("\n");

        }
      /* if ( p->left != NULL) { */

        printf ("    %2d    : ", p -> tostate);

        temp = p->left;
        count = 0;

        while (( temp != NULL)  /* && ( p->stopstate != TRUE) */ ){
            if ( count++ >= 2) {
                printf ("\n          : ");
                count = 0;
            }
            printf ("( %x, %-9s %1s %-5s , %2d )  ",
                    temp,
            /* printf ("( %-9s %1s %-5s , %2d )  ", */
                    spalpha[eventalpha[phase][temp->indx2alpha].spindex],
                    "@",
                    sapalpha[eventalpha[phase][temp->indx2alpha].sapindex],
                    temp -> tostate);
            temp = temp -> right;
        }
      /*}*/

      /* do not forget the stop state */
      if ( p->stopstate == TRUE)
            printf ( " STOP state\n", p ->tostate);
      else {
            printf ("\n");
      }
        printtree(FALSE, p-> left, phase);
        printtree(FALSE, p->right, phase);
    }
    return ;
}
```

Appendix C. Implementation Rules for Relative Clocks [Yu 92]

The following implementation rules apply to relative clocks and are taken from [Yu 92]. For more detailed information please refer to the original reference:

1. Each SAP_i is assigned a counter, C_i, as the relative clock of all user observable events occurring at that SAP. All counters are initialized to "1".
2. Each user increments its counter by 1 when a new local event occurs at its SAP.
3. i) If event "SAP_i.a" is the send if a message "a" through SAP_i, the message a contains the relative timestamp $T(a)=C_i$ at SAP_i.
4. ii) When a message b with relative timestamp T(b) is received at SAP_j from a remote user, counter C_j is incremented as $C_j = 1+Max(T(b),C_j)$.
5. Spontaneous events do not have timestamps.
6. Immediately after a modification of a relative clock according to one of the above implementation rules, the relative time of an event is defined to be the value of the relative clock located at the SAP where it occurs.

Informally, two events occurring at different SAPs are said to be relatively concurrent if either event could possibly occur before the other.

References

[Aho 74] A. Aho, J. E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974

[Boch 80] G. v. Bochmann, *A General Transition Model for Protocols and Communication Services*, IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980

[Boch 89] G. v. Bochmann, *Protocol Specification for OSI*, Computer Networks and ISDN Systems (1989/90) 18: 167-184

[Boyc 89a] T. T.Boyce, T. Grenier, R. L. Probert and H. Ural, *Formalization of ISDN LAPD for Conformance Testing*, Proceedings of the IEEE INFOCOM'89 - The Conference on Computer Communications, April 1989

[Boyc 89b] T. T. Boyce and R. L. Probert, *A Modified Breath First Labelling Algorithm For Functional Decomposition of Communications Protocols*, Technical Report TR-89-22, University of Ottawa, July 1989

[Boyc 90] T. T. Boyce and R. L. Probert, *Phase-Directed Testing of Estelle Specifications*, Proceedings of IFIP International Workshop on Protocol Test Systems, 1990

[Choi 86] T. Y. Choi and R. Miller, *Protocol Analysis and Synthesis by Structured Partitions*, Computer Networks and ISDN Systems (1986)11: 367-381

[Chow 85] C. Chow, M. G. Gouda and S. S. Lam, *A Discipline for Constructing Multiphase Communication Protocols*, ACM Transactions on Computer Systems, vol. 3, No. 4, 1985, pp. 315-343

[ChuL 88a]P.M. Chu and M.T. Liu, *Synthesizing Protocol Specification from Service Specification*, Proceedings of Computer Networking Symposium, April 1988, pp. 173 - 182

[ChuL 88b]P.M. Chu and M.T. Liu, *Protocol Synthesis in a State Transition Model*, Proceedings of COMPSAC 1988, pp. 505 - 512

[Dsou 95]S. Some, R. Dsouli and J. Vaucher, *From Scenarios to Timed Automata: Building Specifications from Users Requirements*, in Proceedings of 2nd Asia Pacific Software Engineering Conference, December 1995

[Faci 90]M. Faci, L. Logrippo and B. Stepien, *Formal Specification of Telephone Systems in LOTOS*, Proceedings of 9th IFIP International Symposium on Protocol Specification, Verification and Testing, 1990

[Faci 91]M. Faci, L. Logrippo and B. Stepien, *Formal Specification of Telephone Systems in LOTOS: Constraint-Oriented Approach*, Computer Networks and ISDN Systems 21 (1991) 53 - 67

[Favr 87]J. P. Favreau and R.J. Linn, *Automatic Generation of Test Scenario Skeletons From Protocol Specifications Written in ESTELLE*, Proceedings of 6th IFIP International Symposium on Protocol Specification, Verification and Testing, 1987

[FORTE]*Proceedings of IFIP TC/WG 6.1 International Conference on Formal Description Techniques for Distributed Communications Protocols*, Elsevier Science Publishers, 1988 - Present

[Hsia 94]P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima and C. Chen, *Formal Approach to Scenario Analysis*, IEEE Software, March 1994

[ISO 86] International Standards Organization, *OSI - Transport Service Definition*, ISO 8072, 1986

[IWPTS] *Proceedings of IFIP TC/WG 6.1 International Workshop on Protocol Test Systems*, Elsevier Science Publishers, 1988 - Present

[Jain 90] B. N. Jain and A. K. Agrawala, *Open Systems Interconnection: Its Architecture and Protocols*, Elsevier Science Publishers, 1990

[Kaku 94] Y. Kakuda, H. Igarashi and T. Kikuno, *Automated Synthesis of Protocol Specifications With Message Collisions and Verification of Timeliness*, Proceedings. 1994 International Conference on Network Protocols, October 1994, pp. 143 - 150

[Koha 78] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Book Company, 1978

[Linn 94] R. J. Linn and M.U. Uyar, *Conformance Testing Methodologies and Architecture for OSI Protocols*, IEEE Computer Society Press, 1994

[Linz 90] P. Linz, *An Introduction to Formal Languages and Automata*, D. C. Heath and Company, 1990

[Mill 90] Raymond E. Miller, *Protocol Verification: The First Ten Years, The Next Ten Years; Some Personal Observations*, Proceedings of 10th IFIP International Symposium on Protocol Specification, Testing and Verification, 1990, *pp. 199-225.*

[Prob 91] R. L. Probert and Kassem Saleh, *Synthesis of Communication Protocols: Survey and Assessment*, IEEE Transactions on Computers, Vol. 40, No. 4, April 1991

[Prob 92] R. L. Probert, H. Yu and K. Saleh, *Relative-Clock-Based Specification and Test Result Analysis of Distributed Systems,* IPCCC 92

[PTSV] *Proceedings of IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Elsevier Science Publishers, 1981 - Present

[Sale 91] Kassem Saleh, *Synthesis Methods for the Design and Validation of Communication Protocols*, Ph.D. thesis, University of Ottawa, January 1991

[Sale 96] K. Saleh, R. L. Probert and I. Manonmani, *Recovery of Communications Protocol Design from Protocol Execution Traces,* in Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems, October 1996

[Sari 93] B. Sarikaya, *Principles of Protocol Engineering and Conformance Testing,* Ellis Harwood Series in Computer Communications, 1993

[Scol 87] G. Scollo and M.V. Sinderen, *On the Architectural Design of the Formal Specification of the Session Standards in LOTOS*, Proceedings of 6th IFIP International Symposium on Protocol Specification, Verification and Testing, 1987

[Yu 92] Hualong Yu, *Testability-Directed Specification of Communications Software,* Master of Computer Science Thesis, University of Ottawa, 1992

[Viss 85] C. Vissers and L. Logrippo, *On the Importance of Service Concept,* Proceedings of 5th IFIP International Symposium on Protocol Specification, Verification and Testing, 1985, pp. 3 - 17

[Viss 88] C. A. Vissers, G. Scollo and M. v. Sinderen, *Architecture and Specification Style in Formal Descriptions of Distributed Systems,* Proceedings of 8th IFIP International Symposium on Protocol Specification, Verification and Testing, 1988

[Viss 93] C. A. Vissers, M. V. Sinderen and L. F. Pires, *What Makes Industries Believe in Formal Methods,* Proceedings of 13th IFIP International Symposium on Protocol Specification, Testing and Verification, 1993