Detecting Feature Interactions in Telecommunications Systems Designs

Mohammed Faci

Thesis Submitted to the School of Graduate Studies and Research
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science
under the Auspices of the Ottawa-Carleton Institute of Computer Science

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA
CARLETON UNIVERSITY

Department of Computer Science,
University of Ottawa,
Ottawa, Ontario, Canada

This research was supported by the Telecommunications Research Institute of Ontario (TRIO).

Abstract

A basic telephone service is used to establish a communication session between two users. A telephone feature, such as *call waiting, call forwarding*, and *three way calling*, is defined as an added functionality of the basic telephone service. A *feature interaction* is the interference of the functionality of one telephone feature with the functionality of another telephone feature, meaning that the invocation of the first feature modifies the functionality of the other active feature, or even prevents its functionality altogether. This problem has become a major obstacle for the extension of telephone systems with new services. Our contributions in this thesis are the following: (1) define a model for specifying telephone systems and their services; (2) formalize the notion of feature interaction; and (3) develop a design methodology for detecting feature interactions at the specification level.

In the first part of the thesis, we define a model for structuring the components of telephone specifications. In this model, a specification is expressed as a set of communicating processes representing three types of constraints: *local constraints, end-to-end constraints,* and *global constraints.* Each of these constraints is defined and used in our specifications.

In the second part of the thesis, we develop a methodology for detecting feature interactions. Central to our methodology is the formalization of the notion of *feature interaction.* Intuitively, we say that an interaction exists between *n* features if one of the features cannot exhibit its behaviour when integrated into POTS in combination with other features. We formalize this concept by defining the *composition* and *integration* of features. *Composition* expresses the synchronization of features on their common actions with POTS and their interleaving on their independent actions. *Integration* expresses the extension of POTS with the *n* features, such that each feature is able to execute all of its actions which are allowed in the context of POTS, when the other features are disabled. Then, we reason about interactions in terms of the *conformance* relation studied in testing theory, in the following way. An interaction exists between *n* features if their *integration* does not *conform* to their *composition*. A set of concrete telephony examples is used to show the applicability of the methodology.

The specification language used is LOTOS, which turns out to be ideally suited to express our methodology. However, our method is general enough to be adapted to other specification languages with similar constructs.

Acknowledgments

Except for errors and omissions, for which I remain fully responsible, the completion of this thesis is due to many people. First, I thank my supervisor Dr. Luigi Logrippo for his friendship, support and guidance. His enthusiasm about research topics and his technical knowledge about telecommunications made my stay at the University of Ottawa a truly pleasant one, and motivated me to pursue research on the most challenging topic that the telephony industry has faced in many years. His careful reading of this thesis improved the content and the form many folds. Luigi thank you. Second, I thank the members of my committee Dr. Richard Carver (external examiner), Dr. Raymond Buhr, Dr. Sunil Das, and Dr. Dan Ionescu for accepting to read this thesis, and for their many suggestions for improving it.

This work was made possible by the financial support of *Telecommunications Research Institute of Ontario*, the *National Institute of Standards and Technology, Bellcore,* and *BNR*.

I want to thank the many people, too numerous to mention explicitly, who gave me the chance to interact with them and learn from them. I apologize to those whose names slip my mind at the moment. As for the rest, I want to thank Bernard Stepien for his collaboration for defining the model discussed in Chapter 4. A note of appreciation goes to my colleague Dr. Mohammed Erradi for allowing me to explore research ideas with him at my convenience and for his comments on an early version of Chapter 5. I also wish to single out Daniel Amyot and Hans van der Schoot for their detailed comments on an advanced version of Chapter 5. Next, I am grateful to Edward Hibon for specifying and integrating the features presented in Chapter 6, which allowed me to validate the theory and describe the examples from a practical point of view. Finally, I want to thank Jacques Sincennes and Antoine Bonavita for their technical support and for their help with the use of the LOTOS toolkit.

A huge thank you goes to my wife Khalida for her crucial support. I want to thank her especially for her encouragements and companionship. I want to thank her for tolerating my evenings and weekends at the office. More importantly, I want to apologize to her for my absent-mindedness when I am physically present. And, of course a million thanks to my 11-months old son Radwan, whose innocent smiles are just as inspiring as the words of wisdom that I hear from my parents all the time. Yes, the list of acknowledgments would not be complete without an explicit mention of my parents, Rekia and Mostefa. I thank them for their unconditional support, for constantly reminding me to *“seek knowledge from the crib to the grave”*, and for trusting me to manage my own life.

Table of Contents

CHAPTER 5 A Methodology for Detecting Feature Interactions

CHAPTER 6 Specifying Features and Detecting their Interactions: Case Studies

List of Figures

CHAPTER 1 Introduction: Motivation and Background

Our work is motivated by the challenges, from a designer's point of view, of detecting interactions between features in a telephone system. Features that perform their functions satisfactorily on their own may, in some instances, be prevented from doing so in the presence of other features. Examples of such effects of one feature on the behaviour of another feature are given in Chapter 6 of this thesis. This problem, which has been termed the *feature interaction problem* [BDCG89], is the subject of this thesis.

Telecommunications software is just as difficult to develop and modify as software in general. The software crisis of the 70's was the tip of the iceberg of new technical challenges, which spanned many fronts of the software life cycle, starting from how to capture user requirements and ending with how to deliver the expected end product on budget and on time. These challenges became even more visible with the rapid advances in new technology and the integration of communication networks and telephone systems. Consequently, the last ten years have become an experimental playground for researchers to develop new software techniques and exploit old ones efficiently.

Software that was built for early networks did not take advantage of some fundamental concepts that are established for software development today. For example, most software was specified using natural languages or the same programming languages that were used for development. As time went by, researchers and developers became aware that development of software products *must* begin by a *precise* formulation of the problem to be solved, if they are to avoid the high cost evolution and maintenance that is usually associated with the end products. For this reason, we witnessed a spur of activities on the international arena, such as the International Organization for Standardization (ISO) and IUT-T (CCITT - International Telegraph and Telephone Consultative Committee), to develop formal specification techniques which can be used both for capturing requirements and for designing systems.

1.1 The Feature Interaction Problem: Where Does this Thesis Fit?

Broadly speaking, the feature interaction problem can be approached from three different angles: detection, avoidance, and resolution [CaVe93] [Fits94]. Furthermore, detection and resolution may be divided into on-line and off-line techniques, as given in the introduction of [Fits94]. Off-line deals with the problem before deployment; on-line deals with it after deployment. Figure 1.1 identifies, as shown by solid lines, the topic of our thesis within this categorization.

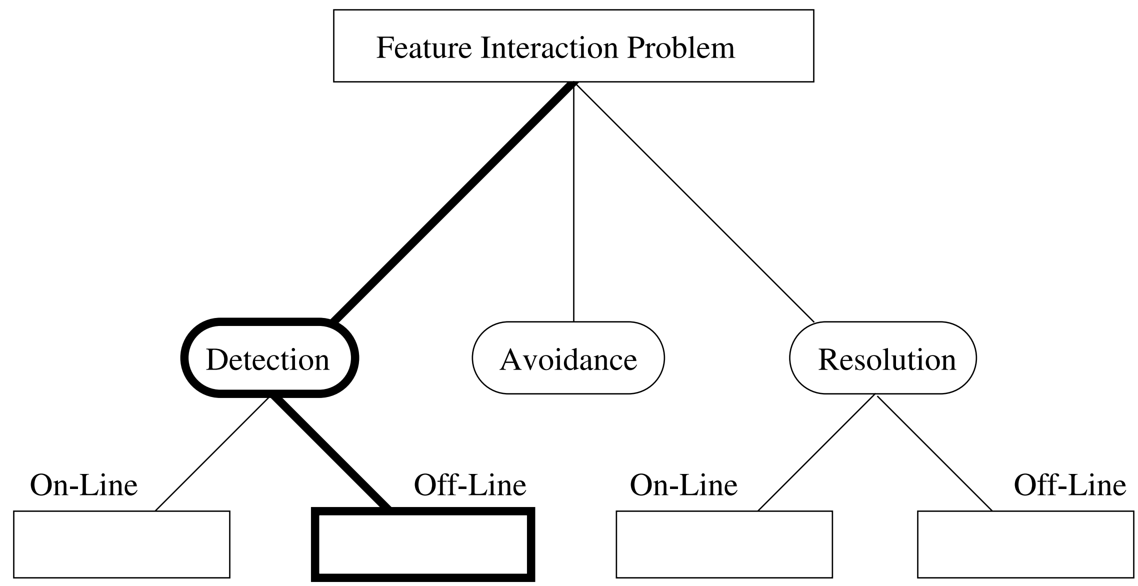


Fig. 1.1 Categories of the Feature Interaction Problem.

The objective of a detection approach is to analyse a set of independently specified features and determine whether or not there are any conflicts between their joint behaviour [CaLi91], [Lee92], [BoLo93], [DaNa93]. An avoidance mechanism assumes that the causes of the interactions are known and an architectural or analytical approach is defined to prevent the manifestation of such interactions [MiTJ93]. The avoidance approach is most suitable in the early phases of specification and design of features. Finally, the objective of a resolution mechanism is to find appropriate solutions to interactions that manifest themselves at execution time. Several approaches have been proposed[Cain92] [GrVe92] [Chen94].

1.2 Contributions of the thesis

As mentioned, this thesis addresses the problem of detecting *feature interactions* in modern telecommunications systems. Our major contribution in this thesis is the development of a methodology for detecting feature interactions in telephone systems [FaLo95], at the specification level. However, some steps of the methodology require the investigation of some fundamental concepts, leading to other contributions. To summarize:

1.2.1 Contribution 1: The Methodology

The methodology calls on the specifier to carry out the following:

- Specify each feature *independently*, within the context of the existing system, using the notion of constraints, as described in contribution 2.
- *Formalize* the notion of feature interaction, then define the terms *composition* and *integration* of features, based on the results of contribution 3.
- Derive a set of test cases, using the theory for the derivation of tests for LOTOS processes, to find cases for which the integration does not *conform* to the composition. This is the subject of contribution 4.

A test case for which the integration does not *conform* to the composition reveals that the way these two features are integrated in the system does not allow for their simultaneous activation, meaning that an interaction exists between them.

1.2.2 Contribution 2: Using Constraints to Specify Telephone Systems

- Use the *constraint-oriented style* to show how to structure the specification of the basic telephone service (POTS). The goal here is that the resulting specification should be independent of implementation.

This idea is developed in Chapter 4. We have categorized constraints, which are used to structure the specification of a telephone system, as *local, end-to-end* and *global*. The notion of constraints allows the designer to express the behaviour of communicating entities from three different views, each complementing the other two. *Local constraints* are used to enforce the appropriate sequences of events within each process, where a process may represent a caller side or a called side. *End-to-End* constraints synchronize the actions of two or more processes, satisfying local constraints, with respect to each other. *Global constraints* are a higher level abstraction than the end-to-end constraints, which are imposed on the global behaviour of the system.

1.2.3 Contribution 3: Formalizing the notion of Feature Interaction

- *Formalize* the notion of feature interaction. This is an important contribution, since it allows us to reason about interactions in terms of a precise definition.

A formal definition of what is meant by feature interaction is presented in Chapter 5. Intuitively, we say that an interaction exists between *n* features if one of the features *cannot exhibit the same behaviour*, when integrated into POTS by itself, as it does when integrated in combination with the other features. In Chapter 5, we formalize the notion of feature interaction, by defining the terms *composition* and *integration*. *Composition* expresses the *simultaneous* execution of features, in the following sense. We say that *n* features execute *simultaneous*ly if they are allowed to synchronize on their common actions with POTS and interleave on their independent actions. *Integration* expresses the result of extending a POTS system with *n* features such that each feature is able to execute all of its traces which are allowed in the context of POTS, when the other features are disabled.

1.2.4 Contribution 4: Relating the LOTOS Testing Theory to Feature Interactions

- Relate the notion of feature interactions to the concept of conformance, as defined for labelled transition systems, in the following way. An interaction exists between *n* features if the *integration* of the features does not **conform** to their *composition*.

Intuitively, a feature interaction occurs when a user invokes a feature expecting some behaviour to occur, but the system behaves otherwise. This mismatch of behaviour between the user's expectation and the system's actual behaviour is expressed as a deadlock in LOTOS, since the user's next action and that of the system are out of synch.

This idea is developed in Chapter 5. We use the testing theory of LOTOS and exploit the results reported in [BrSS87], [Brin88], [BrAL90], we show that the notion of *conformance testing* has a direct application in the domain of detecting feature interactions. Conformance, as defined in the context of Labelled Transition Systems, allows one to reason about two specifications using a single formalism. In our context, the specification representing the *integration (I)* is taken to be an abstract representation of a physical realization and the specification representing the *composition* is taken to be a description of the system's desired behaviour. Informally, to check whether *I conforms to C* corresponds to the following. *I* conforms to *C* if testing *I* against the traces of *C* does not lead to deadlocks that would not occur while testing *C* against those same tests. In other words, testing the integration does not produce deadlocks that would not be discovered while testing the composition.

It is also worth emphasizing that this contribution is limited to the exportation of LOTOS work from the protocols domain to the feature interaction domain. We do not develop new LOTOS theories, rather we develop a novel way of expressing feature interactions in terms of LOTOS deadlocks, using well established results.

1.2.5 Contribution 5: Application of the Methodology to Concrete Telephone Examples

- Apply the methodology to a number of concrete telephone examples.

Chapter 6 shows how to apply the methodology to a number of examples, based on the work of Cameron et al. [CGLN94]. An overview of their work is given in Chapter 2. In order to avoid dealing with the details of specifying switching elements, we limited ourselves to interactions that occur between features of the same user and interactions that occur between features of two different users, independent of the number of switching elements connecting the users.

1.2.6 Related Issues not Addressed in the Thesis

In addition to the specific problem that we intend to address in this thesis, there are many related issues that we will not touch on. In particular, we distinguish between implementation-*dependent* specifications and implementation-*independent* specifications. Our methodology leads to specifications which are implementation-independent. In other words, our specifications have a "logical" structure that may not map directly onto an implementation. In Section 4.1 on page 38, we discuss specification styles and how implementation-independent specifications can be transformed into implementation-dependent ones.

Another important issue that we do not address in the thesis is how to specify telephone systems so that features can be added without having to specify the cross product of existing features and added ones.

Finally, our thesis deals with interactions that occur at the user's level only. This means that we have left out many interesting interactions. One example is the type of interactions which are due to timing (i.e., how long one presses on the flashhook button may mean a flashhook signal or a hang up). Another example is the type of interactions which are due to distributed support of features (i.e., can one bypass his/her own originating call screening list by calling the operator and ask for someone on the screening list?).

1.3 The Seven Chapters of this Thesis

This thesis is structured as follows. Chapter 2 is a literature review. It presents some formalisms that are used to specify telecommunications software, gives some known classifications of the different types of feature interactions, and surveys some of the approaches that have been proposed to solve this problem.

Chapter 3 gives a brief overview of the LOTOS language and justifies its use for specifying and designing telecommunications software.

Chapter 4 describes a model for specifying telephone systems. The formal specification of a moderate size telephone system is presented to show that LOTOS is just as suitable for specifying telephone systems as it is suitable for specifying the communications protocols and services for which it was originally developed.

Chapter 5 addresses the formalization of the notion of feature interaction by giving a precise definition of what it means for a set of features to interact with each other. Next, we define the notions of *integration* and *composition* of features in the context of POTS. Then, we relate the *integration* and *composition* of features to the *conformance* framework, as defined for LOTOS. An interaction is detected between *n* features if their *composition* does not *conform* to their *integration*. In the last part of the Chapter, we apply the methodology to a concrete telephone example: *Call Waiting* and *Three Way Calling*.

Chapter 6 shows the application of the methodology to a collection of pair-wise feature interaction examples. For each example, we give an informal description of the two features, followed by their formal specifications, and finally we derive a testing process to show how to detect the interaction.

Some conclusions and further research directions are given in Chapter 7.

1.4 How to Read this Thesis

We suggest that everyone reads Chapter 2 because it gives a broad overview of some related work. This allows the reader to put our work into perspective. Those who are familiar with LOTOS and the expansion theorem, may glance quickly through Chapter 3, without loss of understanding. Those who have only a general interest in the feature interaction problem, may simply read Chapters 4 and 5, and then skim through Chapter 6. On the other hand, those whose interests are aligned with the subject of this thesis, should read the examples in Chapter 6 in detail. Chapter 7 gives a summary of what was accomplished in the thesis and explores some directions for future research.

CHAPTER 2 Related Work: Formal Methods for Specifying Telephone Features

2.1 Feature Interactions Terminology

The First International Workshop on Feature Interactions in Telecommunications Software Systems[FWFI92] concluded that much research is needed to make some progress towards building better telecommunications systems. It was also concluded that the best place to start with was to develop a framework of definitions and terminology which can be used as the basis for defining research objectives and comparing results. The Second Conference [FWFI94] was more focused, in terms of its technical contents, than its predecessor, but no common terminology or standards emerged from the conference either. So, due to the lack of an established framework at the present time, we define our own terminology that we use throughout this thesis. Our definitions are specific to telecommunications systems.

2.1.1 What is a Service?

In the context of telecommunications, the term service has been used in many different ways. Sometimes [Lin93] it refers to a "complete" package that includes call processing, billing, operations and maintenance (e.g., POTS). Sometimes it refers to tariffable features such as Call Waiting. Yet other times, it refers to a package of features. In this thesis, we will adapt the definition that a service is a marketable unit that consists of a set of features.

2.1.2 Telecommunication Subscriber

A subscriber is an interface entity, external to the system, which subscribes to a telecommunication system in order to obtain a service.

2.1.3 Telecommunication User

A user is an interface entity, external to the system, which uses telecommunication services without necessarily subscribing to them. For example, an individual may use some features of the telephone service, such as the 411 information line (in North America) to obtain a directory number and the emergency 911 distress line to report an emergency. The user need not subscribe to the telephone service in order to use these features.

2.1.4 What is a Feature?

A feature is a functionality of a service. It can be described in terms of some useful behaviour that the user can obtain as part of a telecommunication service. Thus, each feature is a marketable component of a telecommunications service. A feature can be accessed independently or within the context of another feature.

2.1.5 What is a Feature Interaction?

In general terms, a feature interaction is any new behaviour (desirable or undesirable) that results when combining two or more feature behaviours. A more specific and precise definition in terms of our research is given in Definition 5.4 on page 73.

2.1.6 What Do We Mean by Feature Interaction Detection?

In this thesis, the detection of feature interactions means the following. Given a system which has been augmented by a number of features, analyse this system for new behaviours which are not part of any of the individual features or behaviours which are part of the individual features, but cannot be completed in the combined system. A natural next step would be to decide whether such interactions are allowed or not, but this is not considered in this thesis.

2.1.7 What is POTS?

We use POTS, which stands for Plain Old Telephone Service, as our basic system to which we add new features. This means that all interactions are detected with reference to this basic specification.

2.2 Formal Specifications of Telephone Systems

In this section we conduct, with no attempt to be exhaustive, a survey of a number of methods and languages that are used for the specification of telephone systems. A brief description of each technique and how it is used is given.

2.2.1 Finite State Machines

A *finite state machine* (FSM) is an abstract machine that is used to represent the behaviour of a given system in terms of *states* and *transitions*. The most common notation used to represent a FSM is a directed graph whose nodes are system states and whose arcs are system transitions; the other notation being state transition matrices. The machine can be in only one state at a time. Upon receiving an input, the machine generates an output and may change to a new state. Both the output and the new state are functions of the input and the current state. A state is a mean by which one can describe an aspect of the system's behaviour. For example, one may talk about a *Dialing* state, *a Ringing* state, or a *Talking* state while describing the behaviour of a telephone system.

The major problem with a FSM machine is, of course, the state explosion problem. As the number of states increases, the global system state becomes hard to manage. Despite this drawback, several researchers have used FSM to describe telephony applications [Kawa71] [WhCh81]. However, in order to capture systems beyond the most elementary ones, various types of extensions to the basic FSM models have been proposed such as extended FSMs and Communicating FSMs.

Whitis and Chiang [WhCh81] specify a telephone system as a collection of call processing programs, represented as a tree of communicating FSMs. Each FSM performs a specific call processing function. The tree of FSMs is organized into four levels of abstractions: Hardware Interface, Telephone Set Handlers, Virtual Telephone Set Handlers, and Call Managers. The hardware interface is implemented in firmware while the other three layers are implemented in software. Each layer is implemented as a set of FSMs which communicate strictly with upper and lower layers. The objective of their design decision is to maximize the cohesion and to minimize the coupling between the FSMs. The authors also use the notion of *sub-automata* to further structure the system. A sub-automaton is a state machine which can be invoked by other FSMs, similar to procedure calls in conventional programing languages. The authors claim that new features, which are also specified as FSMs, may be introduced into the basic call model by defining an interface between the new features and the model. The interface may require the introduction of additional FSMs to accomplish the integration. However, they do not show how their approach would work on a real application.

2.2.2 SDL

SDL (Specification and Description Language) is the most widely used language in the field of telecommunications[BeHo89]. It has been developed and standardized by CCITT (the International Telegraph and Telephone Consultative Committee). SDL is used to describe both the behaviour and structure of systems, from a high level description down to a detailed design level. The behaviour of a system is described in terms of a set of processes, which is an extended finite state machine. Processes work concurrently and communicate asynchronously with each other by sending and receiving discrete messages called *signal*s. Signals are also the mean by which SDL processes communicate with the environment. When signals are used to communicate between processes, they always carry the identifiers, which are unique, of the sending and receiving processes, along with possible data values.

Some proposals for object oriented SDL have also been put forward [BeMD87] [More89] and are now in the process of being standardized [OSDL92]. Also, since SDL is the most widely used language in the telecommunications industry, many methodologies and support tools have been developed to facilitate its use. For example, an SDL toolbox to support different environments was developed by Nilsson and al. [NiLM89]. A reachability graph generator for SDL specifications is developed by Nguyen et al. [NgJP89] and transformation techniques for translating SDL specifications into implementations is described by Hallsteinsn et al. [HVNM89] and Cheng et al. [ChJa89].

Despite its popular use in developing switching systems [CHCK89], such as the TDX 10, Gelli [Gell87] points out many deficiencies of SDL with respect to LOTOS, our language of choice for the formal specification of telecommunications systems. We select four of those deficiencies. First, LOTOS is more powerful in its capability to describe process communication. Second, LOTOS has a more complete formalism for abstract data types, since it enables data parameterization more than does SDL. Third, because of the synchronous model on which LOTOS is based, system testing, by providing *observer* processes similar to those defined for Estelle, is easier to accomplish. Fourth, because of its formal semantics and explicit theories of observational equivalence, LOTOS offers the best possibilities for checking consistency, completeness, coherence and absence of deadlock.

2.2.3 State Transition Rules

State transition rules (STR)[HiHT90] is a method, based on the finite state machine(FSM) model[Boch78] [Boch80] [BoSu80], which has been developed for the formal specification of telecommunications services. FSMs are augmented with the notion of *state primitives* for representing a large number of states. In the STR method, a process is represented by a set of state primitives. State primitives are divided into four categories: (1) Speech-path description: it has two states, busy or idle; (2) Output description: audible tone or display information; (3) Service memory description: It describes the active set of services. A memory description for a call forward is an example; and (4) Pseudo-primitives: These are related to onhook transition descriptions.

A service is specified, using state primitives, by a set of rules. A rule has the following form: {current-state-description

event-description: next-state-description}

The following example should clarify the concepts of STR:

1.dialtone(A), idle(B)

dial(A, B): ringback(A, B), ringing(B, A).

This rule specifies that if process A is in the dialtone state, process B is in the idle state, and A dials B, then A moves to the ringback state and B moves to the ringing state.

The STR method seems to offer the advantage of ease of use, but it inherits many of the drawbacks that are identified in informal or semiformal methods and languages. First, the global state of the parties involved in a connection is implicit in the rule description. And, second, the method has no formal semantics.

2.2.4 Statecharts

Statecharts[Hare87] [Davi88] are a *visual formalism*, which has been developed for the purpose of specifying complex reactive systems. It extends the FSM model by incorporating the notions of *depth, orthogonality,* and *communication* into the model. Depth, which treats both *clustering* and *refinemen*t, is introduced as a means to express abstraction levels. Refinement is used for top down designs and clustering is used for bottom up designs. For example, a FSM with three states labelled A, B, and C, as shown in Figure 2.1, can be transformed into a FSM with two states only, D and C, by clustering the two states A and B into a single state D. State D is called a *superstate*. So, if *a* is a transition from state A to state C as well as a transition from state B to state C in the original FSM, then transition *a* is also a transition from state D to state C in the new FSM. Conversely, in a top down approach, state D can be refined into two states,

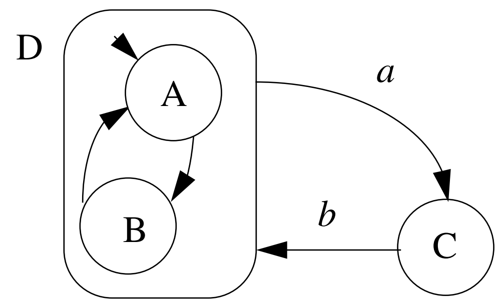


Fig. 2.1 Clustering of States in StateCharts

namely A and B. The implied semantics of this decomposition is that of an *or* function. When the machine is in state D at the higher level of abstraction, it is really either in state A or in state B. A small arrow is used to identify the next state, A in this case, in the subordinate states. Harel also considers the *and* decomposition of superstates into subordinate states. Figure 2.2 on page 13 shows the decomposition of state B into two concurrent states B11 and B12. Concurrency is expressed by dashed lines. When transition *x* occurs, the machine moves to both B11 and B12 simultaneously. Another feature of statecharts is the *conditional transition.* A

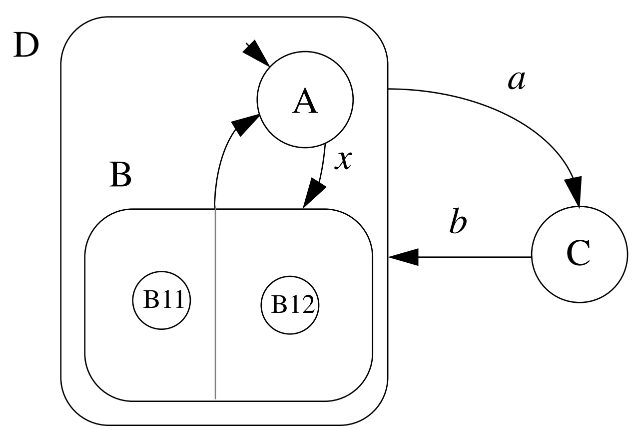


Fig. 2.2 Decomposition of States in StateCharts

FSM may transition from one state to another only if a certain condition holds. A transition may also be specified based on whether the FSM is in a particular state. For example, a FSM representing the sending entity of a telephone specification, may move from state *Idle* to state *Ringing* only if a concurrent FSM, say the receiving entity, is in state *Free.*

Harel claims that this formalism can be used for specifying all types of reactive systems including telephone systems. However, we are not aware of complete specifications of telephone systems in the open literature.

2.2.5 Zave's Technique

In Zave's approach, a telephone system is composed of a set of *views.* A view is basically a partial functionality of the whole system. For example, the system described in [Zave85] is represented in terms of four views: lifetime view, POTS view, Call forward view, and automatic callback view. The views are supervised by a global manager called the *Filter.* A filter receives inputs from the environment and then directs them to the relevant views. While some inputs may be relevant to more than one view, others may not be relevant to any view, in which case an error action is generated by the filter. Each view is described in terms of sequence diagrams, borrowed from Jackson Structured Programming and Jackson System Development. Inputs may also have aliases.

This approach provides a nice graphical formalism which is attractive to systems analysts and developers. However, similar to Jackson's diagrams, it has no rigorous formal semantics basis. In addition, it is difficult to fully evaluate the approach because the example described in [Zave85] does not contain a complete specification.

In a more recent work [ZaJa91], [Zave93], [ZaJa93] Zave and Jackson address the problem of specifying feature-rich telephone systems so as to minimize feature interactions. Starting from the point of view that specifying modern telephone systems requires the use of other approaches, other than single paradigm techniques, they present a multi-paradigms technique for specifying and reasoning about switching systems and their features. Their technique offers the advantage of producing specifications which are easy to reason about, more readable and more concise than specifications written in a single language. With this technique, a typical system contains a *central specification* written in a general-purpose state-oriented language while the set of operations that the system receives are expressed using other languages such as finite-state automata and regular expressions. In some respect, their approach is related to our constraint-oriented specification style. The major difference is that they compose heterogeneous components, whereas we compose LOTOS processes.

2.2.6 Petri Nets

Petri nets[Pete77] are abstract machines which are used to describe the behaviour of systems. They are represented by a directed graph containing two types of elements: places and transitions. Places, which contain tokens, are represented by circles; transitions, which allow tokens to move between places, are represented by lines. Directed graphs connect places to transitions. A transition is said to fire if (a) it is triggered by a clock pulse and (b) all arrows entering the transition originate from places which contain tokens. The Petri-nets model has many followers and it would be long to give a satisfactory account of its many uses.

The Petri-net based model has been used to describe, among other applications[Ager79], the behaviour of telephone switching systems[YoBa77]. Yoeli and Barzalai introduce the concept of extended Petri nets (EPN) and use it to model the call processing operations in an automatic telephone exchange.

In their approach, the telephone system is decomposed into a set of *virtual* subsystems. Four such subsystems are identified: a virtual station subsystem(VSS), a virtual station control(VSC), a virtual dial control (VDC), and a virtual central control(VCC). VSC represents the central exchange. VCD represents the module which collects the dialed digits. The VCC represents the module which handles the establishment of a connection between two users. And finally, VSS represents the user's station. When a user dials a digit, it is transmitted to the VDC, through the VCS subsystem. When the caller manages to reach the caller, the connection is handled by the VCC. The authors illustrate the use of EPN by representing the behaviour of the VSC. The interested reader is encouraged to consult [YoBa77].

2.2.7 Object Oriented Approach

The object oriented delta approach for the specification of telecommunications systems is explored by Simon [Simo89]. Delta, in this context, refers to the fact that the basic telecommunication service, which is specified as a collection of communicating objects, is supplemented by a differential behaviour. This differential behaviour expresses additions or modifications to the basic service behaviour. The delta behaviour is derived from the basic service behaviour using inheritance. Two approaches, the peer supplementary service delta object and the subclass supplementary service delta object, are explored as means to integrate the delta behaviour into the basic service behaviour. A number of services, such as hold, call forwarding, and call waiting, are specified and simulated, using a Multitasking Object-Oriented Prolog (Mlog) programming language[Kara89].

Mlog combines many concepts from object oriented programming, Prolog, and CSP. It supports the concepts of object, meta-class, inheritance, methods, and message passing. Mlog computations are based on the execution paradigm of Prolog, where methods are Prolog procedures and message passing is similar to solving goals. Interprocess communication in Mlog is achieved through the rendezvous mechanism.

Our work differs from that of Simon in several ways. First, the delta specification approach is based on visual formalism, which may lack the preciseness of a formal language such as LOTOS. Second, her approach deals more with the problem of specifying new services with respect to a given basic service, which leaves open (at least from a formal point of view) the question of how to deal with conflicts that arise from attempting to supplement the basic service with two services or more. Our thesis formalizes the notion of service conflicts, which we call *feature interactions*, and defines a methodology for detecting the interactions.

Erradi et al. [EKDB92] also use the object-oriented paradigm for the formal specification of distributed systems. Their environment consists of an object-oriented language called Mondel and a simulator for executing their specifications. They stress the dynamic extension aspects of Mondel specifications. Their approach uses an extension-based algorithm for merging the behaviour of specifications. They claim that telephone features may be specified as independent Mondel objects, which are then used to extend the basic telephone system. Their approach sounds promising for specifying and analysing telephone systems [ErFa94].

2.2.8 University of Ottawa LOTOS group

Our research group has been active in applying LOTOS to the specification of telephone systems. An early study[FaLS90] has shown that LOTOS is well suited for specifying elementary telephone systems, basically the Plain Old Telephone Service(POTS). The results of that study motivated us to explore the application of specification styles [ViSV88] [VSVB91] to the same type of systems. A model for structuring telephone specifications was then developed and applied to the same example. The results were reported in [FaLS91], and reproduced, with minor changes, in Chapter 3 of this thesis. The same model, which we use throughout this thesis, was successfully applied to the specification of more advanced features of telephone systems, such as call forward, ring again, and three way calling [Boum91], [BoLo93] and to the specification of Integrated Services Digital Network (ISDN) services[ErHM92].

2.3 Classifications of Feature Interactions

Existing feature interactions classifications are based on practical experience and observations. To achieve a better understanding of the concepts related to the feature interaction problem, we survey three classifications which are due to Cameron et al. [CGLN94], Cain [Cain92], and Inoue et al. [Inoue92].

Cameron et al. [CGLN94] have categorized feature interactions by the *nature* of interactions and by the *cause* on the interactions. Categorization by the nature of interactions is defined in terms of three elements: the *kind of Features* involved; the *number of users* involved; and the *number of network elements* involved. Within the context of the kinds of features, two kinds are recognized: *customer features* and *system features.* Customer features include all of the call processing features visible to the general public such as *call waiting* and *call forwarding*. System features include system functions such as billing, administration and maintenance. Within the context of number of users, the distinction is made between *single users* and *multiple users.* Single-user interactions occur when different features of the same user influence the functionality of each other; multiple-user interactions occur when features active on different connections interfere with the functionality of each other. Finally, within the context of the number of elements involved, the distinction is made between *single-element* interactions and *multiple-element* interactions. An element refers to a transmission line, a telephone set, a display unit, or a switch. *Single-element* interactions occur when only a single network element is involved in handling call processing. *Multiple-element* interactions occur when features are being supported, simultaneously, by several network elements. Due to the distributed nature of call processing systems, multiple-element interactions is becoming very common.

Combining the number of customers, with access to certain features, and the number of elements supporting those feature, the following combination of interactions results: single-user-single-element, single-user-multiple-element, multiple-user-single-element and multiple-user-multiple-element. In this thesis, we will abstract from the aspect of elements and detect the interactions between the features of the same user and the features of two different users, independent of the number of switching elements.

Categorization by the cause of the interaction is defined in terms of 3 elements. First, *violations of assumptions about the system.* These include violations of assumptions about naming, data availability, administrative domain, call control and signaling protocol. Second, *Limitations on network support*. These include limited customer premises equipment signaling capabilities and limited functionalities for communications among network elements. Finally, *problems of large distributed systems*. These include personalized instantiation of feature parameters, timing and race conditions, distributed support of features and non-atomic operations. Readers interested in this categorization may consult [CGLN94] for further details.

Cain[Cain92] describes three high level relationships for specifying the interactions between features. These relationships are: Mutual exclusion, Priority, and Dependency. The mutual exclusion restricts the processing of an event to a single active feature at a time. Clearly, if features are activated in a mutual exclusion fashion, feature interactions are a priori prevented. The priority relationship imposes an ordering on a set of concurrently active features, if they require the processing of a shared event. By doing so, feature interactions are prevented by directing an event, which is of interest to many active features, to the highest priority feature. And finally, dependency. If the activation of a feature is dependent on the activation of another feature, then the order of activation, to respond to a common event, must be specified.

Although Cain presents an interesting approach of managing interactions, he does not define a methodology for actually detecting interactions, which is the focus of our thesis.

Inoue et al. [Inoue92] classify feature interaction into two types: non-deterministic state transitions and lack of appropriate state transitions. They also propose an approach for automatic detection and elimination of these two types of interactions. We give further details about their detection approach in the next section.

2.4 Approaches for Detecting Feature Interactions

In this section we give a brief overview of some approaches for detecting feature interactions, with a particular interest to (1) how to specify features (2) how to define the notion of feature interaction, and (3) how to detect feature interactions. Only research directly related to our work is discussed.

Braithwaite and Atlee [BrAt94] model telephone systems as layered (similar to a stack) state-transition machines, using a tabular notation. The basic telephone service, which is described in terms of the originating and the terminating call model, is at the bottom of the stack. The stack grows as features are added to it. Information, which is represented as *tokens*, propagates from the top of the stack to the basic telephone service and back up again. Tokens are intercepted by each feature, which may modify them or simply pass them on to the next layer.

Their detection approach is based on the informal definition for the notion of feature interaction; in their context, a "feature interaction occurs when one feature affects the behaviour of another". To detect an interaction between two features, the composition of all possible call configurations is obtained and the resulting reachability graph is explored for interactions, which have the following interpretations with respect to the graph: conflicts between the desired information in a state and information contained in that state, relationships between assertions in a particular state, and perturbation in the paths of a component machine when considered in the global composition. Their approach has been used to detect specific classes of interactions such as call control interactions, resource contentions, information invalidations, and assertion invalidations.

Another approach for detecting feature (service) interactions is described by Ohta and Harada [OhHa94]. The basic telephone service is seen as a black box; new services are added to the system, based on the observable behaviour of the basic service. Services are specified in STR, as introduced in Section 2.2.3 on page 11. Since STR is based on FSMs, the analysis of feature interactions is driven by the analysis of the combined transition rules, with respect to the global states, that are relevant to all the features under consideration. Although no explicit definition of the term feature interaction is given, their approach for composing features and analysing them is conceptually similar to ours. Feature interactions are explained in terms of specific problems, referred to as *logical* and *semantical*, that can be detected in the global composed behaviour of the features. Logical problems include deadlock states, states from which no transition is possible, and *nondeterministic* states, states from which more than one transition is possible for the same event. Semantical problems are categorized as illegal transitions, lost transitions, illegal states, lost states, and duplicate definitions of terminology.

Next, we describe the work of Lin and Lin [LiLi94]. Their informal definition of the term feature interaction is close to the one we use, but no formal definition is given either: "Detecting interactions can be viewed as a process of checking whether the actual behaviours of a feature, in the presence of other features, are different from the intended behaviours of the feature". The detection of feature interactions is based on a 5-step approach. The first step defines the two Basic Call Models (BCMs), which are related to the originating side and the terminating side of a connection. The second step uses the BCMs to construct three Basic Feature Contexts (BFCs), which are the originating BFC, the terminating BFC, and the two-party BCF. The third step describes how to combine BFCs with a feature's logic to model the operations of the feature. Features are described at two levels: procedural, which is a sequence of computational steps and behavioural, which is a set of functional properties. The procedural specification represents the actual behaviours of a feature, and the behavioural specification captures the intended behaviour. The fourth step describes the composition of the features in a single context. The different feature contexts are merged together to yield a single composed context; interactions between the features is seen as a violation of the individual properties of certain building blocks in the composite context. The final step describes the use of a *feature manager*, which is responsible for resolving interactions among features.

Conceptually, their approach is similar to ours. Their two-level approach for describing features can be easily mapped into our local and global constraints. End-to-end constraints are more explicit in our model than in theirs. Finally, we both use the notion of *context*, which has an important role in the design of features with respect to their environment. An important difference between the two approaches is the formalization of the notion of feature interaction, which is central to our methodology.

Another interesting research is that of Lee [Lee92], where he explores the object-oriented approach using Object-Z for the specification and analysis of telecommunications services. The main goal of this work is to demonstrate the use of formal specifications for the purpose of analysing the interactions that result from the combination of telecommunication services. Object-Z is used to specify the basic telephone system, which can then be extended to include additional services. They derive several examples which offer the basic telephone service and one additional service. Interaction analyses are carried out on a small combination of these services. The conclusion of their work is that the large number of possible combinations of potential telecommunications services results in such a complexity that no general approach for analysing their interactions appears to be possible. Although we use a different approach and define a formal framework for detecting feature interactions, we agree with their main conclusion in that the problem is complex, and it may be worth investigating other approaches such as the multi-paradigms technique [ZaJa94].

CHAPTER 3 An Overview of LOTOS

In order to make the thesis self-contained, this Chapter introduces the reader to LOTOS (*Language Of Temporal Ordering Specifications*) from an operational point of view. It follows closely the tutorial of Logrippo et al. [LoFH92] in format and spirit. The reader is referred to Bolognesi and Brinksma's paper [BoBr87] for the theoretical foundations of the language, such as formal definitions of LOTOS concepts, inference rules and behavioral equivalences. Related work on the language can be found in [ISO88a], [Tur88b], [ISO89], [VaVD89], [ISO90].

3.1 Mathematical Concepts and Notations

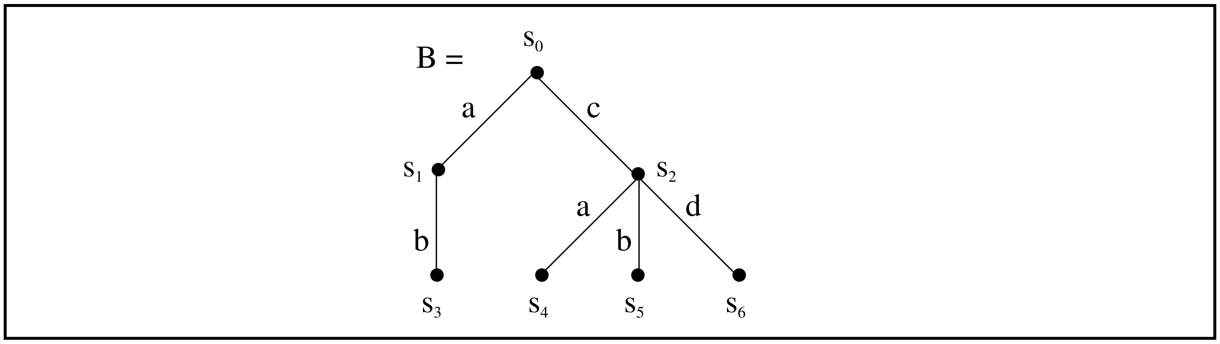
LOTOS, as has already been mentioned, is based on the concept of *labeled transition systems* (LTS's). LTS's are a generalization of finite state machines and provide a convenient way for expressing the step-by-step operational semantics of processes. Processes evolve by executing one action at a time, selected from their alphabet set. Formally [BoBr87],

Definition 3.1: Labeled Transition System

A *labeled transition system (lts)* is a 4-tuple LTS = $<S, s_0, L, T>$ where,

- S is a (finite) non-empty set of states;
- s_0 in S is the initial state;
- L is a (finite) set of observable actions; and
- $T = \{ \xrightarrow{a} \subseteq S \times S \mid a \in L' \text{, where } L' = L \cup \{i\}\}$, the *transitions*, is a set of binary relations on S. So, if $s_1 \xrightarrow{a} s_2$ such that $s_1, s_2 \in S$ then $<s_1, s_2> \in \xrightarrow{a}$.

Here is an example of a behaviour B represented by an LTS which happens to be a *labeled transition tree*.



The interpretation of this tree is as follows:

- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$;
- $L = \{a, b, c, d\}$ and $L' = \{a, b, c, d, i\}$;
- $T = \{<s_0, a, s_1>, <s_1, b, s_3>, <s_0, c, s_2>, <s_2, a, s_4>, <s_2, b, s_5>, <s_2, d, s_6>\}$;
- $s_0 = s_0$.

The following notations and definitions are widely used for interpreting LTS's [DeHe84, Brin88, Ledu92].

- $L = \{a, b, c, ...\}$ is the alphabet of observable actions and i is the hidden action;
- $B -a\rightarrow B'$ means that after executing the observable action a, the behaviour expression B is transformed into another behaviour expression B';
- $B - i^k \rightarrow B'$ means that after executing a sequence of k hidden actions, the behaviour expression B is transformed into another behaviour expression B';
- $B -ab\rightarrow B'$ means that $\exists$ B'' such that: $B - a \rightarrow B''$ **and** $B'' - b \rightarrow B'$;
- $B =a\Rightarrow B'$ means that B is transformed into another behaviour expression B' by executing zero or more internal actions, followed by the observable action a, then zero or more internal actions. Formally, $\exists\ k_0, k_1 \in N,$ such that $B - i^{k_0}\ a\ i^{k_1}\rightarrow B'$;
- $B =a\Rightarrow$ means that B may accept the action a. Formally, $\exists$ B': $B =a \Rightarrow B'$;
- $B \neq a\Rightarrow B'$ means **not**($B = a \Rightarrow B'$), that is B, **must** refuse the action a;
- $B =\sigma\Rightarrow B'$ means that B is transformed into another behaviour expression B' by executing a sequence of observable actions. Formally, if $\sigma = a_1...a_n$ then $\exists\ k_0, ..., k_n \in N$: $B - i^{k_0}\ a_1\ i^{k_1}\ a_2\ \dots\ a_n\ i^{k_n} \rightarrow B'$;
- $B =\sigma\Rightarrow$ means that $\exists$ B': $B = \sigma \Rightarrow B'$;
- B *after* $\sigma = \{B' \mid B = \sigma \Rightarrow B'\}$, i.e., the set of all behaviour expressions reachable from B after executing the sequence σ;

- A trace is a sequence of actions; $t \sqsubseteq t'$ expresses the fact that t is a, not necessarily contiguous, *subtrace* of t';
- The trace set of B is defined as: $Trace(B) = \{\sigma \mid B = \sigma \Rightarrow\}$. Note that $Trace(B) \subseteq L^*$.

3.2 Overview of LOTOS

LOTOS [ISO88a] (Language Of Temporal Ordering Specifications) is a by-product of the effort of standardization of the Open Systems Interconnection (OSI) within the International Organization for Standardization (ISO). The main desirable characteristics of LOTOS are: abstractness, implementation-independence, formal semantics and support of verification methods. LOTOS has two parts. The behaviour part is based mostly on Milner's Calculus of Communicating Systems (CCS) [Miln89] and Hoare's Communicating Sequential Processes (CSP) [Hoar85], and the data part is based on Ehrig and Mahr's ACT ONE [EhMa85].

3.2.1 LOTOS Principles

Some of the principles that have inspired the design of LOTOS are:

1. *Formal definition:* Formally defined syntax, static semantics, and dynamic semantics. In particular, the dynamic semantics of LOTOS are described operationally in terms of inference rules [BoBr87].
2. *Process algebra:* Following Milner's ideas, the operational semantics are defined in such a way that it is possible to prove a rich set of algebraic equivalence properties, based on several types of equivalence relations. These properties can be used in order to prove equivalence or correctness of specifications, as well as to transform the structure of a specification.
3. *Interleaving concurrency:* Events are considered to be atomic, and thus the parallel execution of two events *a* and *b* is defined as a situation of choice, where *a* can occur before *b,* or vice versa. Therefore, any LOTOS behavior expression can be rewritten as an expression consisting of a choice between behavior expressions, each prefixed by a single action (i.e., expansion theorem [Miln80], [Miln89]). This principle will be used in our thesis, where we explain the operators by examples for which we show the expansion.

4. *Executability:* Because LOTOS semantics are defined operationally, it is possible to implement these semantics in an interpreter, which for a behavior expression can enumerate the set of possible next actions [LOBF88][Vane88], and the behavior expressions resulting by the execution of each one of them (this is another application of the expansion theorem). Although this set can be infinite, in many cases it can still be described in finite terms. This means that LOTOS specifications can be written, without difficulty, in such a way as to be interpretable, or even, with some user-supplied information, to be translatable into a program [MaMi89].
5. *Modularity and module reusability:* LOTOS favors stepwise decomposition of processes. By using parameterization, these processes become reusable.

Much of the power of LOTOS is the result of the power of the parallel composition operator, and of the concept of process *rendezvous*, which is called process *synchronization*. Their semantics are quite different from those found in most common programming languages. The following are some of the salient characteristics of these concepts:

1. *Multiway synchronization:* While much of LOTOS semantics are based on CCS, the multiway synchronization concept was borrowed from Hoare's CSP. In order for synchronization to occur, a number of processes may have to participate, as will be described below.
2. *Symmetric synchronization:* As mentioned above, all processes that participate in a synchronization cooperate in it equally, in particular there is no concept of a process initiating the synchronization and others responding. For example, while it is often thought that in an output operation, information is transferred from an active process to an external environment, in LOTOS one says that both the process and the environment participate in establishing the value being output. There is no directionality in this concept, although some processes may have more information on the value to be established than others.

3. *Anonymous synchronization:* A process that is ready for a synchronization proposes the synchronization to its environment, without being able to direct its proposal to a specific process. It is the structure of the system where the process is embedded that decides which processes will have to participate for a synchronization to occur. Process identification can be specified as an exchange of appropriate values.
4. *Nondeterministic synchronization.* Often more than one synchronization is possible. One only will be executed according to a nondeterministic choice.

A *behavior expression* represents a state of a process. A predefined set of operators is used to combine actions and behavior expressions to form other behavior expressions. There are two predefined behavior expressions, **stop**, which denotes *deadlock* (or *inaction*) and ***exit***, which denotes *successful termination*. LOTOS *process definitions* are named behavior expressions, similar to procedures in a programming language. Process instantiation is similar to procedure call.

The behavior expression of a process determines which actions (or events) are possible as next actions of the process. There are actions that a process can execute independently, these are represented by the internal action ***i***. And there are actions that need synchronization with the *environment* in order to be executed. These are *offered* at synchronization points called *gates*. The environment of a process consists of other processes, or some external (i.e. non-LOTOS) world that can be a human observer. When an action is executed, the behavior expression of the process is transformed into another behavior expression. It is the inference rules of the dynamic semantics that determine which actions can be offered and executed by a process and how behavior expressions are transformed by effect of the execution of actions. For instance, we shall see that the behavior expression *a; B* evolves into behavior expression *B* after executing action *a*. Similarly, *a; A [] b; B* evolves into behavior expression *A* after executing action *a*.

3.2.2 LOTOS Operators

The syntax of a typical LOTOS specification is given in Figure 3.1. The behavioural part is expressed in a tree like structure, where the root of the tree is the initial state of the system. From this state, the behaviour may execute one of the possible actions, in cooperation with the environment, and move to another state. A process which moves to a state where no actions are possible is expressed by the LOTOS operator **stop**. Other LOTOS operators are explained next.

Specification SpecName [$g_1, g_2, ..., g_n$] ($p_1, p_2, ..., p_m$): functionality
<data type definitions>
behaviour
<behaviour expression>
where
<process definition>
endspec

Fig. 3.1 Syntax of a LOTOS specification

3.2.2.1 The Action Prefix Operator

The *action prefix* operator, written as a semi-colon **;**, expresses sequential composition of actions. This operator is used to sequentially order actions. For example, *a; B* denotes a behavior where action *a* must be executed before the behavior expression *B*.

3.2.2.2 Choice Operator

The choice operator is used to express a choice between two alternatives. A selection between the two alternatives is said to be nondeterministic if the environment cannot distinguish between them. Otherwise, the choice is said to be deterministic. For example, compare the behaviors of three vending machines:

insert_quarter; get_coffee; **stop** [] insert_dime; get_milk; **stop** (* S_1 *)

insert_quarter; get_coffee; **stop** [] insert_quarter; get_milk; **stop** (* S_2 *)

insert_quarter; (get_coffee; **stop** [] get_milk; **stop**) (* S_3 *)

Machine S_1 offers the client (the environment) a choice between inserting a quarter and inserting a dime. If the environment offers a quarter, the behavior of the machine evolves to *get_coffee;* ***stop***. If it offers *insert_dime* it evolves to *get_milk;* ***stop***. Machine S_2 accepts *quarters* only, and after synchronization with the environment (i.e., once the client inserts a quarter), the behavior of the machine can evolve to either *get_coffee;* ***stop*** or *get_milk;* ***stop*** depending on a *nondeterministic choice* (in other words, once the client inserts a *quarter* the choice to synchronize on either *get_coffee* or *get_milk* would no longer exist). Machine S_3 accepts *quarters* only as well, but it is more democratic. Once the client inserts a *quarter*, the behavior of the machine evolves to *(get_coffee;* ***stop*** *[] get_milk;* ***stop****)*, meaning that the client can still choose between coffee and milk. Another way to express nondeterminism in LOTOS is by using the internal action ***i***, to be introduced later.

Nondeterminism is a powerful abstraction mechanism, since it allows to postpone decisions about details that are not relevant at a higher level of abstraction. An implementation of S_2 will have to decide under what conditions each of the two choices is taken, however for specification purposes such a decision may be unessential.

3.2.2.3 Enable Operator

The LOTOS *enable* operator ***>>*** has a similar function as the action prefix operator, which expresses the sequential composition of an action with a behavior expression. The ***>>*** is used to express the sequential composition of two behavior expressions. For example, if *P1* and *P2* are two processes, *P1 > > P2* is read *P1 enables P*2. Process *P1* must terminate successfully in order for *P2* to be enabled. This is the only condition under which process *P2* is enabled. Execution of an ***exit*** in *P1* results in an action on a special gate d. The enable causes d to become an ***i***, and execution to continue with *P2*. For example,

a; b; **exit** >> c; **stop** is equivalent to *a; b;* **i**; *c;* ***stop***, i.e., *a; b; c;* ***stop***,

whereas,

a; b; **stop** >> c; **stop** is equivalent to *a; b;* ***stop***.

because the expression on the right hand side of the enable operator cannot be executed.
In general, we would like to pass parameters from P1 to P2 using the following form:

Let B = B1 >> B2,
where B1 = ...; **exit** (v1, v2, ...,vn) and
B2 = **accept** p1: S1, ..., pn: Sn **in** B3

then B2 is enabled with parameters p1, p2, ...,pn whose values correspond to v1, v2, ...,vn.

3.2.2.4 Internal Action

Being able to be executed independently of the environment, internal actions provide an additional way of specifying nondeterminism in LOTOS.

coffee; **exit** [] milk; **exit**(*1*)

is a process that is ready to synchronize on both *coffee* and *milk*.

i; coffee; **exit** [] milk; **exit**(*2*)

is a process that is ready to synchronize on *coffee*, but may not be able to synchronize on *milk*. If the environment proposes *milk,* synchronization may be impossible if the process has already decided to execute ***i***; however, if the environment proposes *coffee*, it can be assumed that the internal action will be executed eventually, and synchronization will then occur. Similarly,

i; coffee; **exit** [] **i**; milk; **exit**(*3*)

is a process that may be unable to synchronize on either *coffee* or *milk*, depending on an internal decision.

An interesting application of this concept is the specification of priorities. (*2*) can be interpreted that *coffee* has priority over *milk*. If, in addition, one wants to specify that *milk* is still possible after *coffee*, this could be written as:

i; coffee; milk; **exit** [] milk; **exit**

By using more complicated cascades of alternatives with internal actions, one can specify priorities with respect to any predetermined and finite number of events.

3.2.2.5 Parallel Composition

There are three parallel composition operators in LOTOS: a basic one, the *selective composition* operator, and two derived ones, the *interleaving* and the *full synchronization* operators.

The *interleaving* operator (|||) is used to express the concept of parallelism between behaviors when no synchronization is required.

(out1; out2; **exit**) **|||** (in1; in2; **exit**)

is equivalent to (recall the discussion regarding the *interleaving concurrency* model in section 3.2.1):

```
out1;  (out2;    in1;   in2;   exit
        []
        in1;     (out2;    in2 ;   exit   []  in2;  out2;  exit)  )
[]
in1;   (in2;     out1;  out2;  exit
        []
        out1;    (in2;     out2;   exit   []  out2;  in2;   exit  )
```

When processes must synchronize on common actions, the *selective parallel* operator, denoted by *|[L]|*, is used, where *L* is the list of actions on which synchronization must occur. For example:

```
  a; b; exit        (* sub-expression 1 *)
|[a]|
  d;  a; c; exit    (* sub-expression 2 *)
```

is equivalent to

d; a; (b; **exit** ||| c; **exit**)

which is equivalent to

d; a; (b; c; **exit** [] c; b; **exit**)

This is so because both behavior sub-expressions execute independently until one of the sub-expressions reaches a common action, in which case it must wait for the other sub-expression. Once the second sub-expression reaches the same point, synchronization is possible (depending on the context in which the expression occurs, for example participation of other processes may be necessary) and both sub-expressions proceed to offer their next actions. In this example, the first action of sub-expression 1 is *a*. Since action *a* is common to both sub-expressions, it must wait for sub-expression 2 to reach action *a*, before offering action *b*. On the other hand, action *d* of sub-expression 2 is not common, so no synchronization is required. Then they both continue independently, thus sub-expression 1 can offer action *b* and sub-expression 2 can offer action *c*, which can occur in either order.

Note that an action is not treated as common between processes composed in parallel unless it is shown in the list of gates in the parallel composition operator. Clearly, when *L* is the empty list the selective parallel composition operator becomes identical to the interleave operator.

A special case is provided by the action δ, which is produced by ***exit**s*. The action δ is always considered to be a common gate, for any parallel composition operator. Therefore, all behaviors composed in parallel must synchronize on their ***exits***. In the case of enable, the action δ is transformed into an internal action, after the synchronization with other ***exit**s* has occurred. For example,

a; **exit** ||| b; c; **exit**

is equivalent to

a; b; c; **exit** [] b; (a; c; **exit** [] c; a; **exit**)

Both of these behaviors are ready to synchronize with the environment on any of the following sequences:

a b c d

b a c d

b c a d

Let's take another example:

(a; **exit** ||| b; **exit**) >> c; **stop**

is equivalent to

a; b; i; c; **stop** [] b; a; i; c; **stop**

after synchronization on d which has been transformed into *i* by the enable operator. The behavior *c; **stop*** is enabled only after the two ***exits*** synchronize.

The *full synchronization* operator, denoted ||, is used when the processes involved in synchronization must synchronize on every observable action. Clearly, when *L* is the list of all gates, the selective parallel composition operator becomes identical to the full synchronization operator. For example, the behavior: *a; b; c; **exit*** can only synchronize with the behavior: *a; b; c; **exit***. Therefore, *a; b; c; **exit*** || *a; b; c; **exit*** is equivalent to *a; b; c; **exit***. As a second example, the behavior *a; b; **exit*** || *d; a; c; **exit*** is equivalent to ***stop*** (deadlock!), because the left hand side offers *a* while the right hand side offers *d*. However,

a; b; **exit** || (a; c; **exit** [] a; b; **exit**) is equivalent to

a; **stop** [] a; b; **exit**

in other words it can lead to either deadlock or success, depending on a nondeterministic choice: if synchronization occurs on the first *a*, further synchronization is impossible.

In the presence of a choice, all the alternatives that lead to a deadlock are not considered. This can be expressed by the law: *B [] **stop*** is equivalent to *B*. For example,

a; b; **exit** || (a; c; **exit** [] c; b; **exit**)

is equivalent to *a;* ***stop***. The first alternative was selected because the second would have led to immediate deadlock, however the deadlock occurred after the first action.

As a further example, note that

(a; b; **stop** [] c; d; **stop**) |[a,b]| (a; b; **stop** [] d; f; **stop**)

is equivalent to

a; b; **stop** [] (c; d; **stop** ||| d; f; **stop**)

It is also interesting to consider the role of the internal action in this respect. For example,

a; **exit** || (a; **exit** [] i; b; **exit**)

is equivalent to

a; **exit** [] i; **stop**

i.e., it may deadlock if the system chooses to execute the internal action before agreeing on *a*. On the other hand,

(a; **exit** [] b; **exit**) || (a; **exit** [] i; b; **exit**)

is equivalent to

a; **exit** [] i; b; **exit**

i.e. it will not deadlock (if the environment cooperates), while

a; **exit** || (i; a; **exit** [] i; b; **exit**)

is equivalent to

i; a; **exit** [] i; **stop**

i.e. it may deadlock depending on what internal action is executed.

These examples show that, as already mentioned, there is only interleaving on internal actions.

The parallel composition operators *///* and *//* are commutative and associative. *|[L]|* is commutative, and may be associative depending on the gates in *L*.

3.2.2.6 Disable operator

The LOTOS *disable* operator *[>* models an interruption of a process by another process. So, ***P1 [> P2*** means that, at any point during the execution of ***P1***, there is a choice between executing the next action from ***P1*** or the first action from ***P2***. Once the first action from ***P2*** is chosen, ***P2*** continues executing, and ***P1*** is no longer possible. However, if ***P1*** terminates successfully, then ***P2*** does not start execution. For example,

a; b; **exit** [> c; d; **stop**

is equivalent to

a; (b;(**exit**[]c; d; **stop**) [] c; d; **stop**) [] c; d; **stop**

3.2.2.7 Hiding operator

The ***hide*** operator allows abstracting from the internal functioning of a process, by hiding actions that are internal to it. In particular, when the top-down approach is used, the designer can compose the system using LOTOS operators while hiding the details of interprocess communications that are irrelevant at a higher level of abstraction. Here is an example of the ***hide*** operator:

(**hide** b **in** a; b; c; **exit**) || a; c; **exit**

is equivalent to *a;* ***i****; c;* ***exit*** because *b* is turned into an internal action which does not synchronize. Also,

hide b **in** (a; b; c; **exit** || a; c; **exit**)

is equivalent to *a;* ***stop***.

3.3 Full LOTOS

Full LOTOS adds to what we have seen the capability to express data. Although we make extensive use of full LOTOS in our specifications, we abstract from much of its details in presenting our results in this thesis. Therefore, our overview of full LOTOS is kept intentionally brief. Also, no description for ACT ONE [EhMa85] is given.

In full LOTOS it is possible to describe process synchronization involving the exchange of data values. Data structures and value expressions are defined by using the abstract data type (ADT) specification language ACT ONE.

LOTOS variables are basically value identifiers, i.e., place holders for value expressions to be generated during execution. The definition of a variable in LOTOS has the form *VarName*: *Sort* where *VarName* is a variable name that can take any value expression of sort *Sort*. For example, *Caller*: *DecDigit* is a variable whose values are in the domain of *DecDigit*. In order to use the value identifier *Caller*, the data type *DecDigit* (called sorts in LOTOS) must be defined in the ADTs part of the specification.

3.3.1 Actions in full LOTOS

In basic LOTOS, we defined an action to be synonym with gate. In full LOTOS an action is formed of three components: a gate, a list of events, and an optional predicate. Processes synchronize their actions, provided that they name the same gate, that the lists of events are matched, and that the predicates, if any, are satisfied. An event can either offer (!) or accept (?) a value. Predicates establish a condition on the values that can be accepted/offered. As an example, consider the following action:

g ?*Called*: *DecDigit* !*RingsFrom* [*Caller* ***NotIn*** *BusyList*]

This is a LOTOS action that occurs at gate g and expects from the environment a value for *Caller*, offers the value *RingsFrom*, provided that *Caller* is not in the busy list, where ***NotIn*** is an operation on sets, defined in the ADTs part.

3.3.2 Synchronization with value establishment

Interprocess synchronization with value exchange occurs when two or more processes agree on all the values associated with a gate. The following example shows a possible synchronization between an action offered by a Caller process and an action offered by a basic telephone network process.

Caller process:	Pots !*Caller* !*Offhook*; **stop**
	|[Pots]|
Network process:	Pots ?*Caller*: *DecDigit* ?*Offhook*: *Signal*; **stop**

These two processes synchronize with each other, since they have the same gate Pots, and after synchronization, the values *Caller* and *Offhook* are passed from the Caller process to the Network process.

3.3.3 Successful Termination with parameters

This is denoted by exit(e_1, ... ,e_n). The results of the process that executes the exit are passed to the enabled behaviour, if any. The following example shows how the dialling phase may terminate by offering values, such as *Caller* and *Called*, which can be passed to the next phase (ringing phase).

Caller process: Pots !*Caller* !*Dials* ?*Called*: *DecDigit*;
exit (*Caller*, *Called*)

3.3.4 Process Instantiation

A instantiation is an invocation of an already defined process. Actual gates and values are used to instantiate the formal gates and parameters of the process definition. Recursion can be achieved by having a process instantiate itself. For example, given the process definition:

$P[fg_1, ..., fg_n](p_1: s_1, ..., p_n: s_n)$,

the expression $P[ag_1, ..., ag_n](v_1, ..., v_n)$ behaves like P with the actual gates $ag_1, ..., ag_n$ renaming the formal gates $fg_1, ..., fg_n$ and the actual values $v_1, ..., v_n$ replacing the formal parameters $p_1, ..., p_n$.

3.3.5 Guarded behavior

A behavior expression can be preceded by a guard that must be true in order for the expression to be enabled. For example, a Called process in a basic telephone network can only ring if it is not in the busy list. This can be expressed by:

[*Called* ***NotIn*** *BusyList*] -> Pots !*Called* !*Ringfrom* !*Caller*; **stop**

3.3.6 Sequential Composition with Value Passing (enable with accept)

A process may pass values to another process by using the enable construct, which has the form: B1 **>> accept** x_1: S_1, ... ,x_n: S_n **in** B2

B1 will be executed until it terminates. If it exits, the values x_1 to x_n are passed to behaviour B2. The number and sorts of the values that are passed at the successful termination of B1 must be the same as those of the variable declarations in the accept statement. The enable produces an internal action as discussed in Figure 3.2.2.3 on page 27. Let us complete the example given in Section 3.3.3 on page 34.

Let B1 := Pots !*Caller* !*Dials* ?*Called*: DecDigit;
exit (*Caller*, *Called*)

and

B2 := Pots !*User2* !*RingsFrom* !*User1*;
stop

then, in the expression:

B1 **>> accept** *User1*, *User2*: DecDigit **in** B2

B1 passes two values *Caller* and *Called* to process B2. These values are renamed *User1* and *User2* in B2.

3.4 Telephone Architecture and LOTOS Concepts

Expressing the telephone architecture in terms of LOTOS begins by relating the concepts of its architecture to the constructs of the language. Therefore, our first task is to give our definitions of the following terms: system, behaviour, and interaction. Attaching a precise meaning to these concepts is the key to a specification which can be reasoned about, analyzed and validated. Several interpretation of architectural concepts in terms of LOTOS have been put forward [Schot90] [ISO90], some of which are in the process of being standardized [ISO90]. In this section, we identify the relevant telephone concepts in terms of LOTOS.

3.4.1 System

To quote Cassandras [Cass93] "...system is one of those primitive concepts whose understanding might best be left to intuition rather than an exact definition. Nonetheless, [one] can provide the following definition: A system is a combination of components that act together to perform a function not possible with any of the individual parts". In our telephony context, a system is a set of components which can be used to perform the well-known telephony services, of which the most essential one is usually voice transmission. The observer plays the role of the environment which interacts with the system. In this thesis, we represent systems as LOTOS processes.

3.4.2 Behaviour

The system's behaviour is the composed behaviour of its components. Each component, which is expressed as a LOTOS process, exhibits a partial behaviour.

3.4.3 Interaction

System interaction points are represented as LOTOS gates. Processes interact with each other in order to synchronize their actions and exchange information. Processes may exchange simple data values or complex data structures, which are defined as abstract data types. One form of information exchange is achieved through the mechanism of synchronization. Synchronization is also used to coordinate the progress of processes without data exchange. Another form of information exchange is achieved through data parameters passing.

CHAPTER 4 A Model for Specifying Telephone Systems and their Features

A predictable observation that we made at the beginning of our work was that several concepts originally developed for the description of data communications protocols and services are also very appropriate for the description of telephone systems (TSs). One such concept is that of service provider [ViLo86]. In a service specification, only the external behavior of the system is captured, that is, describing *what* the system does for the user, not *how* it does it.

There are, however, important differences between specifying TSs and their related features and specifying OSI services, especially of the type that have been specified in LOTOS so far. For example, more than OSI service specifications, TSs specifications are dominated by the connection and disconnection phases, where the connection is established or released and a number of housekeeping functions is performed, such as the functions of maintaining the list of busy numbers, of checking whether the telephone is in use or not, and of generating busy signals.

An element that cannot be dealt with completely in LOTOS at the present stage is timing aspects. For example, it would be impossible to exactly portray a specification element such as: "the telephone can only be off hook for a maximum of 20 seconds, after which it will be disconnected", although a similar result might be obtained by specifying disconnection after an unspecified amount of time. Some research has been done in this area, and some proposals exist [QuFe87] [QuAF90] [BoLT90] [VaTZ90] [Ledu92]. We are looking forward to progress in this important issue. However, from a pragmatic point of view it should be recognized that timed LOTOS will be a more complex language than the existing LOTOS, and that for many purposes it is not necessary to formally specify exact time delays. In fact, all cases of feature interactions that we consider in this thesis can be represented without time.

4.1 Issues of Specification Style for Telephone System Specifications

Vissers, Scollo, van Sinderen, and Brinksma [ViSV88] [VSVB91] identify four main styles for writing LOTOS specifications, viz. the monolithic style, the state-oriented style, the resource-oriented style and the constraint-oriented style. Each one of these styles has its own uses in TS specifications, and they can be mixed (although of course arbitrary mixture can be counterproductive).

The monolithic style gives explicitly all possible sequences of actions allowed by a specification. The main operator is *choice* [], and the specification is shown as a tree of choices. Therefore, this style is useful for debugging the specification and generating test sequences. A well-known basic result in LOTOS theory is the expansion theorem [Miln89] [ISO88a], by which any LOTOS specification can be transformed into a (possibly infinite!) monolithic one. Although expansion may not terminate, it can yield finite initial subtrees of an infinite monolithic specification equivalent to the given one. Such subtrees can be used for the purposes mentioned above. This "partial expansion" process can be carried out by an interpreter (see the "symbolic execution trees" of [GuHL88][GuLo89]) or by specialized tools [QuPF89]. Symbolic execution trees were used in the design process of our specification.

In the state-oriented style, explicit system states are identified, e.g. by using state variables. Although the state-oriented style is quite tedious if used throughout a specification, it may occasionally be useful to introduce state variables that identify the state of some devices. This may lead to increased readability of the specification in cases where the informal specification uses the state concept, as is quite common for telephone devices. It may also lead to LOTOS specifications which can be implemented directly. In this work, we avoided the state-oriented style because we wanted to produce an abstract specification, without explicit reference to device states.

In the resource-oriented style the processes are chosen in such a way as to represent resources, which means implementation modules. This style is useful for implementation specification.

If it is desired to produce an implementation of the specifications presented in this work, it would be appropriate to translate them into some combination of resource-oriented and/or state-oriented style.

The constraint-oriented style is the most abstract specification style, since it focuses on event sequencing and logical constraints as seen from the external interaction points. In this style, processes identify constraints and usually have little or no relation with implementation processes. This style is, therefore, useful for implementation-independent specifications [Turn87]. One can identify two main types of constraints: *event sequence* constraints and *value* constraints. The former are expressed by mechanisms of *pure* LOTOS, while the latter require values as well. The two main operators used to express process (i.e. constraint) composition in our specification are ||| and ||, i.e. the *interleave* and *synchronization* operators. That is, *(A|||B)||C* means that *A* and *B* can freely interleave, while synchronizing with *C*. For example:

(a; b; **stop** ||| c; d; **stop**)
||
(a; d; **stop** ||| c; b; **stop**)

is equivalent to (in monolithic style)

(a; c; (b; d; **stop**
    []
    d; b; **stop**)
)
[]
(c; a; (b; d; **stop**
    []
        d; b; **stop**)
)

where the latter expression is the *expansion* of the former.

4.2 A Specification of the Plain Old Telephone Service (POTS)

In this section, we present the design method of our specification, explain its structure, and discuss our solutions to some key design decisions. But first, an informal description of POTS is in order. It should be recalled at this point that we are aiming at an abstract service specification. In no way did we attempt to specify a real-life signalling system. It is possible to produce a LOTOS specification of a concrete signalling system, but it would be too complex for our research purposes.

4.2.1 The Informal Description

First, we introduce the terminology that will be used throughout this thesis. *System* refers to the local telephone branch exchange. A *telephone* has two aspects: a *Caller* side and *Called* side. Each telephone is identified by a unique number, which we call the *telephone number* (or simply *number*). A number is associated with a *handset* which the user picks up for dialing another user, or answering a ring. Intuitively, telephone users communicate with each other through a black box (POTS). They interact with POTS by using a well defined set of primitives.

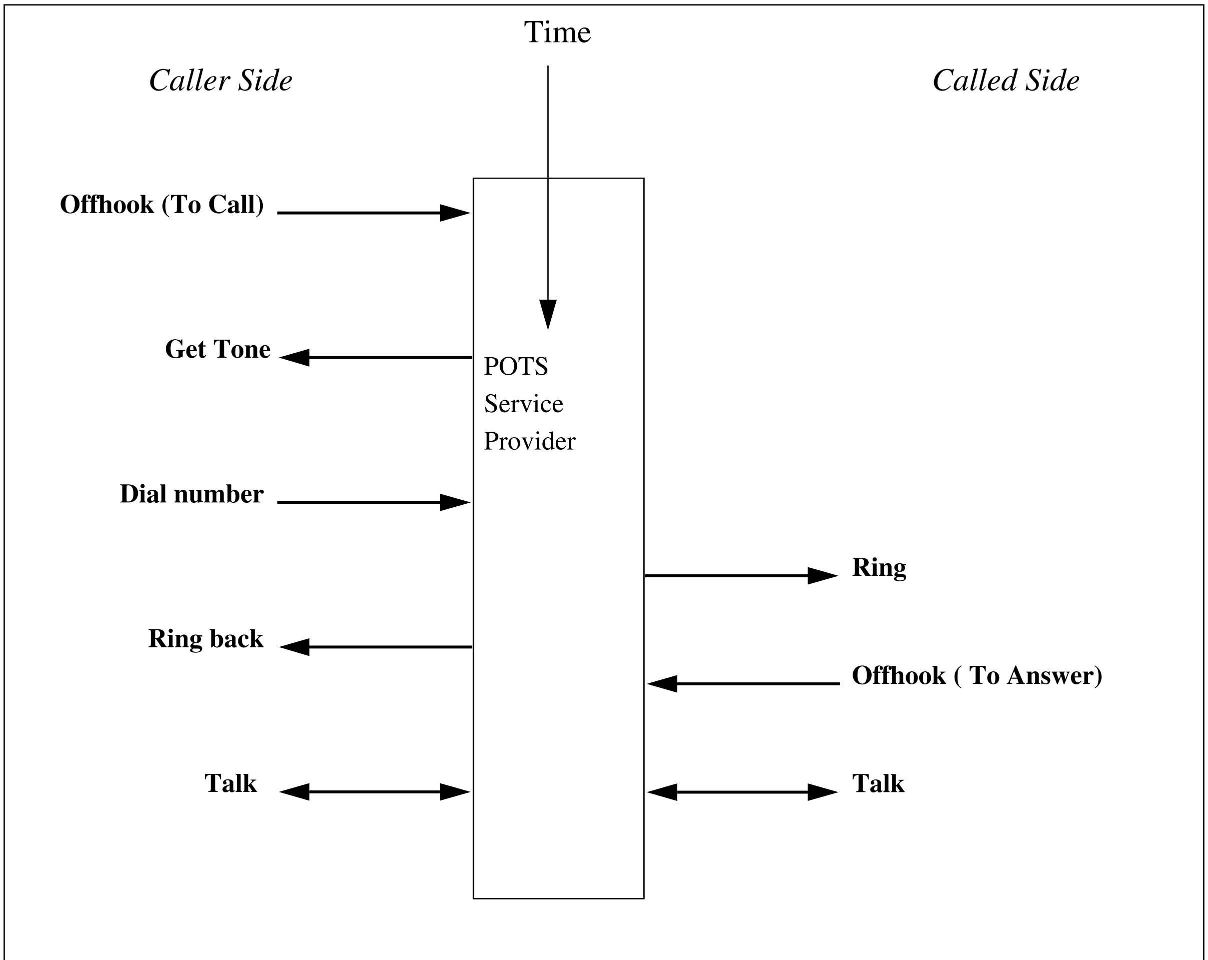


Fig. 4.1 Informal description of a connection between two POTS users

The diagram of Figure 4.1 reflects a scenario where the caller establishes a connection with a non-busy called. This is the kind of diagram that can be found to be useful as a first step in the design process. The diagram is not complete, at this stage, because *hang ups* and *busy signals* are not shown. They will be added in the next design stage.

For further refinement of this diagram we add more information, resulting in a timing diagram, similar to the shown in Figure 4.2, that shows all possible interactions, from a user's point of view. In this specification, only two users per connection are considered.

```
1  Caller Offhook
|  1  Caller GetsTone
|  |  1  Caller Dials Called
|  |  |  1  Caller GetsBusySignal
|  |  |  |  1  Caller HangsUp
|  |  |  2  Caller HangsUp
|  |  |  3  Called Rings
|  |  |  |  1  Caller HangsUp
|  |  |  |  2  Called Offhook
|  |  |  |  |  1  Caller TalksTo Called
|  |  |  |  |  2  Called TalksTo Caller
|  |  |  |  |  3  Caller HangsUp
|  |  |  |  |  4  Called HangsUp
|  |  2  Caller HangsUp
|  2  Caller HangsUp


|  |  |  |  |  |
----------------------------->  Time
```

Fig. 4.2 Sequence of events for the POTS Specification

Our specification allows for an arbitrary number of users to access the TS and communicate with each other. The TS works in the following manner. The first user, the *Caller*, picks up the handset. If no other extension for the same number is in use, the network responds by sending a *tone* signal to the user. The *Caller* is now in a position to *dial* the number of the second user, the *Called*. When the *Caller* completes dialling the number, the telephone network checks if the *Called* number is free, and if so, a *Ring* signal is sent to the second telephone. Otherwise, a *BusySignal* is sent to the *Caller.* If the *Called* user does not pick up the handset to answer, the *Caller* will eventually hang up and both telephones are free. However, if the *Called* picks up the handset, the telephone stops ringing and the two parties engage in a conversation. When the conversation is finished, either party may hang up. The first user who hangs up makes his/her telephone free to make or receive other calls. The second telephone remains busy until the user hangs up.

4.2.2 Structure of the Specification

Figure 4.3 shows the general structure of the LOTOS specifications that we use throughout this thesis. In this figure, we identify *processes* and *gates*. We represent a process of a caller side and a called side by icons of common household telephone sets. We represent a controller process between a caller side and a called side by large white squares. LOTOS gates are identified by small black rectangles. Processes and gates are identified by unique names. Synchronization between the system's components is shown by solid lines connecting the components and the gates which are part of the synchronization set. Note that these lines, as well as the small black squares, represent synchronization points and not channels with capacity as in asynchronous languages such as SDL. This notation is used in subsequent chapters as well.

The figure shows the structure of a single connection between two users **A** and **B**. Processes **A** and **B** interleave with each other and synchronize with the controller process *PotsC*. Gate *Suser* allows a three way synchronization between each of the users, the controller, and the global constraints process while the gate *talk* allows a four way synchronization between the same four processes because, intuitively, the talking phase requires all four processes to participate in the "talk" action. The connection and disconnection phases require, at any given time, the participation of only one of the two communicating sides and the controller. The composition of these three processes is labelled *Connection*. We also show the *Global Constraints* process in the figure, which synchronizes with the *Connection* process on every action. To make the figures simpler, the *Global Constraints* will not be shown, but it is always implicit in the structure.

4.2.3 Behaviour of the Specification

LOTOS makes it possible to describe the behavior of systems in a stepwise fashion, moving from one abstraction level to another. This is a powerful technique since, at each level, it is possible to describe the level's architecture completely. Using this method within the framework of the constraint-oriented specification style, we were able to identify three types of constraints:

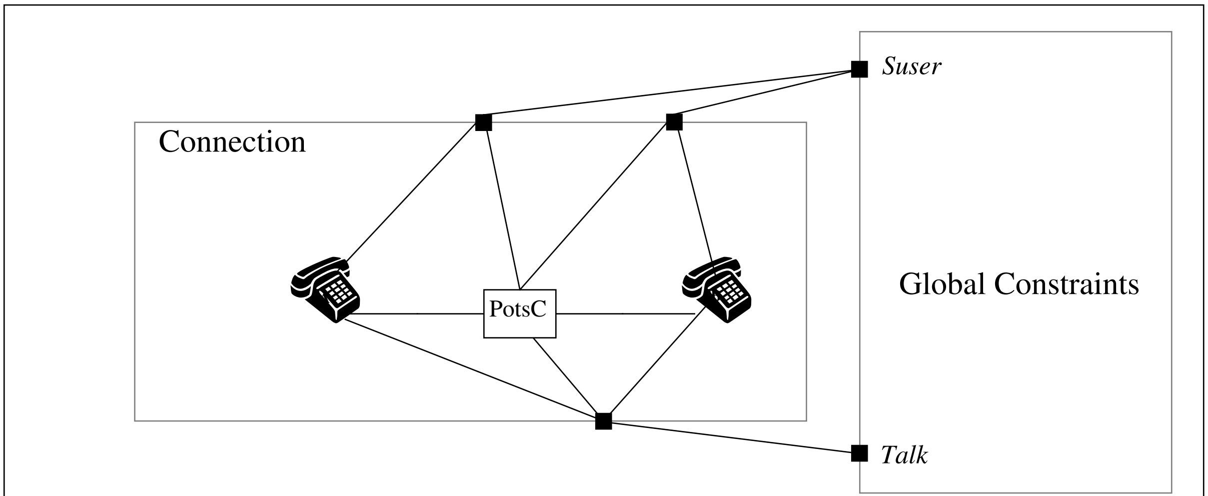


Fig. 4.3 Structure of a POTS Specification using our Model

(1) *Local constraints* are used to enforce the appropriate sequences of events at each telephone, and are different according to whether the telephone is a *Caller* or a *Called*. Therefore local constraints are represented by the processes *Caller* and *Called* and an instance of each of these is associated with each telephone existing in the system. Because these two processes are independent of each other, they are composed by the interleaving operator |||. Another possible choice at this level is to specify only one telephone process, provided with a parameter indicating a *Caller* or a *Called* role for each instantiation [ISO88b].

(2) *End-to-End* constraints are related to each connection, and enforce the appropriate sequence of actions between telephones in a connection. For example, ringing at the *Called* must necessarily follow dialling at the *Caller*. Process *Controller* enforces these constraints. Because they must apply to both *Caller* and *Called, w*e have the structure *(Caller* ||| *Called)* || *Controller*. Thus the controller must participate in every action of the *Caller*, as well as in every action of the *Called,* separately.

(3) *Global constraints* are system-wide constraints. In our specification we identified one main *value* constraint, which is the fact that at any time, a number is used at most once. This constraint is enforced by the process *GlobalConstraints*. Because global constraints must be satisfied simultaneously over the whole system, represented by process *SystemConnections,* we have the structure *SystemConnections* || *GlobalConstraints.*

The notions of local, end-to-end and global constraints are well-known in the area of service specifications [Boch80][ISO88b][ViSV88]. As in [ISO88b], the top level architecture of the specification is obtained by composing in parallel the processes representing the constraints.

The top levels of the LOTOS specification are shown in Figure 4.4. A graphical representation is shown in Figure 4.5. Intuitively, interleaved processes are drawn on top of each other, such as *Caller* and *Called*; parallel processes are drawn next to each other, such as *GlobalConstraints* and *SystemConnections*; and subprocesses are drawn inside the processes by which they are invoked. Dashed lines represent parentheses.

The top-level behavior is composed of two processes, *SystemConnections* and *GlobalConstraints*. These two processes synchronize through gate *Suser. S*tated informally, we want to create as many connections as desired provided that neither the calling nor the called number is already in use. *GlobalConstraints*, which we will describe later, enforces global constraints by keeping track of free and busy numbers and synchronizing with *SystemConnections* to exchange values.

Process *SystemConnections* is composed of two processes: *SingleConnection* interleaved with the process *SystemConnections* itself. This creates the desired effect of being able to have an arbitrary number of connections existing simultaneously. Note the action ***i*** before the recursive call to *SystemConnections*. This can be taken to mean that the creation of a new connection follows internal actions by the system, such as allocation of necessary resources (more technically, it should be considered that if no internal action was specified at this point, the recursion would be unguarded [Miln89], which would make the specification impossible to execute on a simulator). The process *SingleConnection* is viewed as the composition of three processes: *Caller*, *Called* and *Controller*. The conceptual notion of modeling the call initiator (*Caller*) side and the *Called* side by two interleaved processes is quite natural; it reflects the distributed nature of the architecture, in that local constraints apply to separate portions of behavior. *Caller* and *Called* exchange information by synchronization with the *Controller.*

It would be possible to specify an upper bound on the number of possible simultaneous connections, for example by using an additional *counter* and appropriate guards.

Specialists involved in the implementation and simulation of TS might at first be taken aback by this abstract structure. We should emphasize again that our primary objective is to produce a clear and concise specification of the service provided, as made possible by the

```
behaviour
(           SystemConnections[Suser, Talk](InsertL(2, InsertL(5, InsertL(7, {})))) (* Users list *)
            ||
            GlobalView [Suser, Talk]        (    {} of DecSet,        (* Busy users *)
                                                 {} of SetOfPairs     (* enganged set *)
                                            )
)
where
            process SystemConnections[Suser, Talk](UsersList: List):noexit:=
            (
                  PotsConnections [Suser, Talk](UsersList)
            )
            where       process PotsConnections [Suser, Talk](Calling: List): noexit :=
                        (     [Card(Calling) eq 0 of Nat] -> stop
                              []
                              [Card(Calling) ne 0 of Nat] ->
                              (     SingleConnection [Suser, Talk] (First(Calling))
                                    |||
                                    i; PotsConnections[Suser,Talk](RemoveL(First(Calling), Calling))
                              )
                        )
                        endproc
                        process SingleConnection [Suser, Talk] (Caller: DecDigit, CalledUsers: List)
                        : exit (Nat) :=
                        (     (     CallerHandler [Suser, Talk] (Caller)
                                    |[Talk]|
                                    CalledHandler[Suser, Talk]
                              )
                              ||
                              Controller [Suser, Talk] (Caller)
                        )     >>    accept Success: Nat in
                                          SingleConnection [Suser, Talk] (Caller, CalledUsers)
                        endproc
                        process CallerHandler [Suser, Talk] (Caller: DecDigit) : exit (Nat) := ...
                        process CallerHandler [Suser, Talk] : exit (Nat) := ...
```

Fig. 4.4 Top level structure of the POTS Specification

characteristics of the specification language, and no attempt is made to reflect a possible implementation architecture. A different specification style will have to be used for that purpose. And, the resulting specification will be larger and less clear than the one provided here, one of the reasons being that it will have to be dependent on a specific implementation architecture.

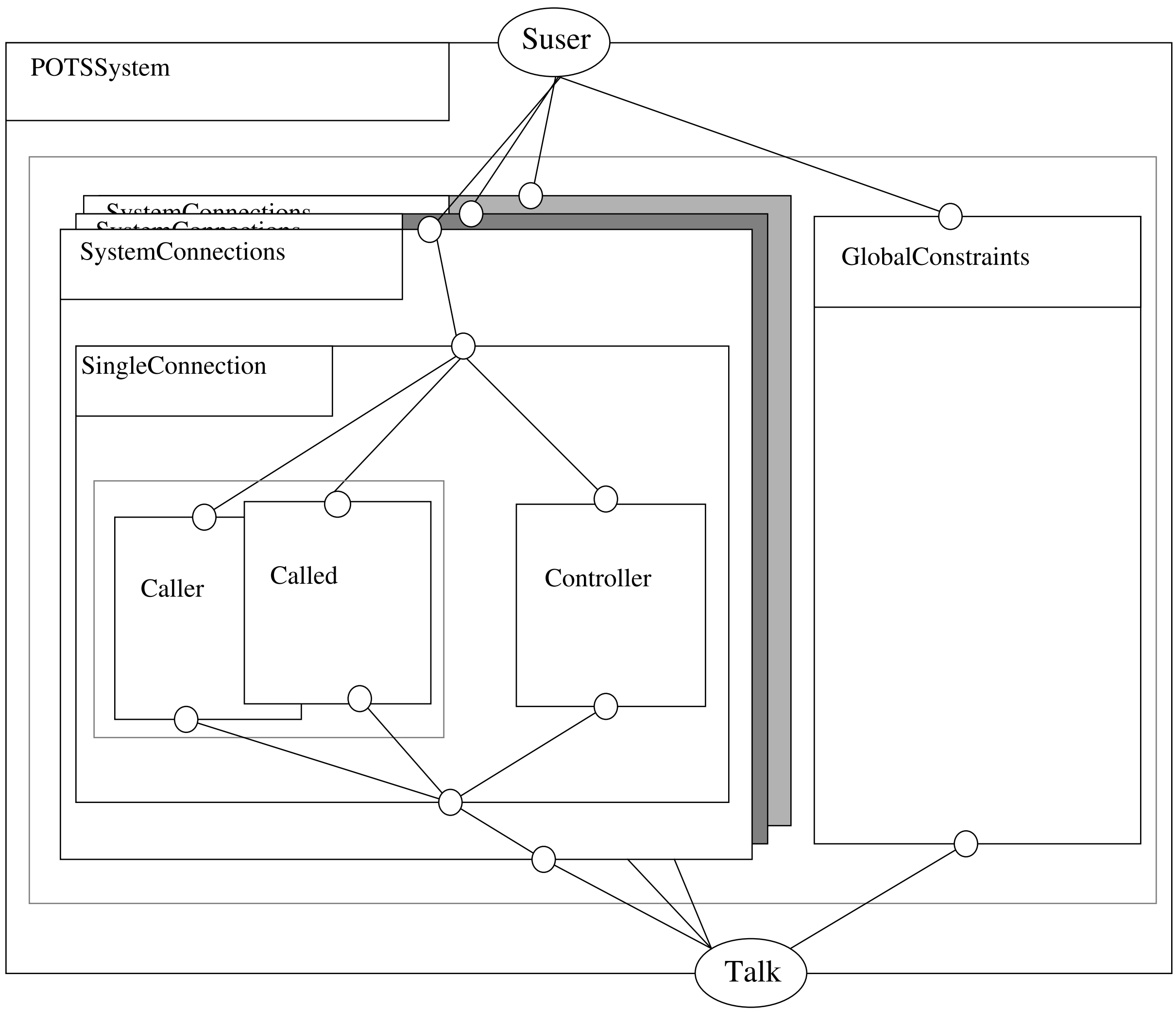


Fig. 4.5 A graphical representation of the specification's top levels

4.2.4 Descriptions of Processes

In this section we first give the details of the four most important processes which make up the specification. And then we present the abstract data types used to maintain the system users. As mentioned earlier, our specification is composed of the following processes: *Caller, Called, Controller,* and *GlobalConstraints.* It is important for the reader to keep in mind that every action, except the ***i***, is a multiway rendezvous, on the gate *Suser*, of three behaviors:

- (*Caller* ||| *Called*),
- *Controller*, and
- *GlobalConstraints*.

Our specification is a model of the switch, where the *Caller* and *Called* processes represent the user who initiates the call and the user who responds to the call, respectively. The *Controller* process, which is also a model of a component in the switch, represents the sequencing of actions between the caller and the called.

4.2.4.1 SystemConnections

The global constraints are enforced by the composition of two processes, *GlobalConstraints* and *SystemConnections*, shown in the top level of the specification's structure. *GlobalConstraints* has exclusive use of the users' phone numbers. It interacts with *SystemConnections* to enforce the sequence of events within each connection as well as the use of the system values across all the active connections. Each connection in the system is composed of three subprocesses *(Caller, Called* and *Controller)*.

4.2.4.2 Caller Side

```
process CallerPots [Suser, Talk](Caller: DecDigit):exit:=
(     Suser !Caller !offhookcall;
      (           (     Suser !Caller !getstone ;
                        Suser !Caller !dials ?Called: DecDigit;
                        (
                              Suser !Caller !ringsback ?Called: DecDigit;
                              UserTalks [Talk](Caller, Called)
                              []
                              CallerGetsBusySignal [Suser](Caller)
                        )
                  )
                  [> UserHangsUp [Suser](Caller)
      )     )
endproc (* CallerPots *)
```

Fig. 4.6 Specification of caller side in POTS

Local constraints are expressed as a sequence of events that must take place locally, either at the *Caller* side or the *Called* side. Given the set of observable actions: *offhookcall, getstone, dials, TalksTo,* and *Busy*, the behaviour of the *Caller* is specified so that only valid sequences are executed for each instance of the *Caller*. For example, the *Caller* cannot dial a number before the tone is received. In addition, the event *hangsup*, may disrupt anywhere after the event *offhookcall* has occurred, The LOTOS code is shown in Figure 4.6.

4.2.4.3 Called Side

At the other end of the connection, the *Called* process is responsible for enforcing its local constraints using another set of actions, which consists of *Rings, offHookans*, and *TalksTo*. Again, *TalksTo* may only be offered after *offhookans* has occurred. Also, the *hangsup* event may occur only after *the offhookans* has occurred. The corresponding LOTOS code is shown in Figure 4.7.

```
process CalledPots [Suser, Talk] : exit :=
(       Suser ?Called: DecDigit !ringsfrom ?Caller: DecDigit;
        (       Suser !Called !offhookans !Caller;
                (       UserTalks [Talk](Caller, Called)
                        [> UserHangsUp [Suser](Called)
                )
                []
                exit
        )
        []
        exit
)
endproc (* CalledPots *)
```

Fig. 4.7 Specification of called side in POTS

4.2.4.4 Controller

The role of the *Controller* is to enforce the end-to-end constraints. The *Controller* combines the independent sequences of the *Caller* and the *Called* into a single ordered sequence of events. For example one such sequence would be: *offhookcall, getstone, dials, ringsfrom, ringsback, offhookans*, followed by *TalksTo.* An abstract representation of the *Controller* with respect to the two sides is shown in Figure 4.8.

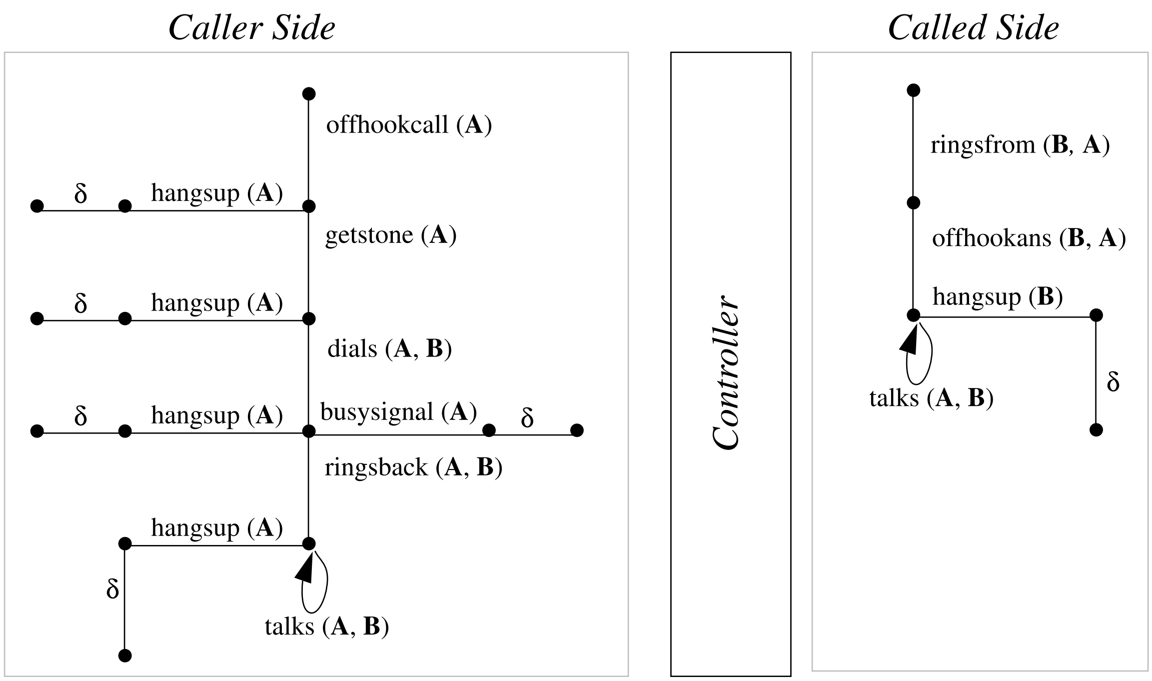


Fig. 4.8 Representation of *Controller* with respect to *Caller* and *Called*

Starting from the initial state, and for all subsequent states, the *Controller* synchronizes either with the caller side or the called side. An interesting ramification of the *hangsup* event is that the *Controller* must deal with it according to its source as well as its point of occurrence. Because of the recursion in the system, a deadlock may be unintentionally introduced if this event is not handled properly. By properly, we mean that the connection must be returned to its initial state regardless of whether the hang up occurred on one side only or on both sides. As Figure 4.9 shows, proper synchronization of the controller with the two sides ensures that the connection terminates successfully. In our specification, we distinguish three cases of the hang up event. These cases are discussed in section 4.3.5, in conjunction with the data structure.

Note that in our LTSs representations, we abstract from LOTOS gates and concentrate on the signals which are exchanged between the users and the system. For example, the LOTOS actions: **user** !*Caller* !*offhookcall* in the *CallerPots* process (Figure 4.6) is represented as *offhookcall(**A**)* in the LTS. This notation is used throughout this thesis.

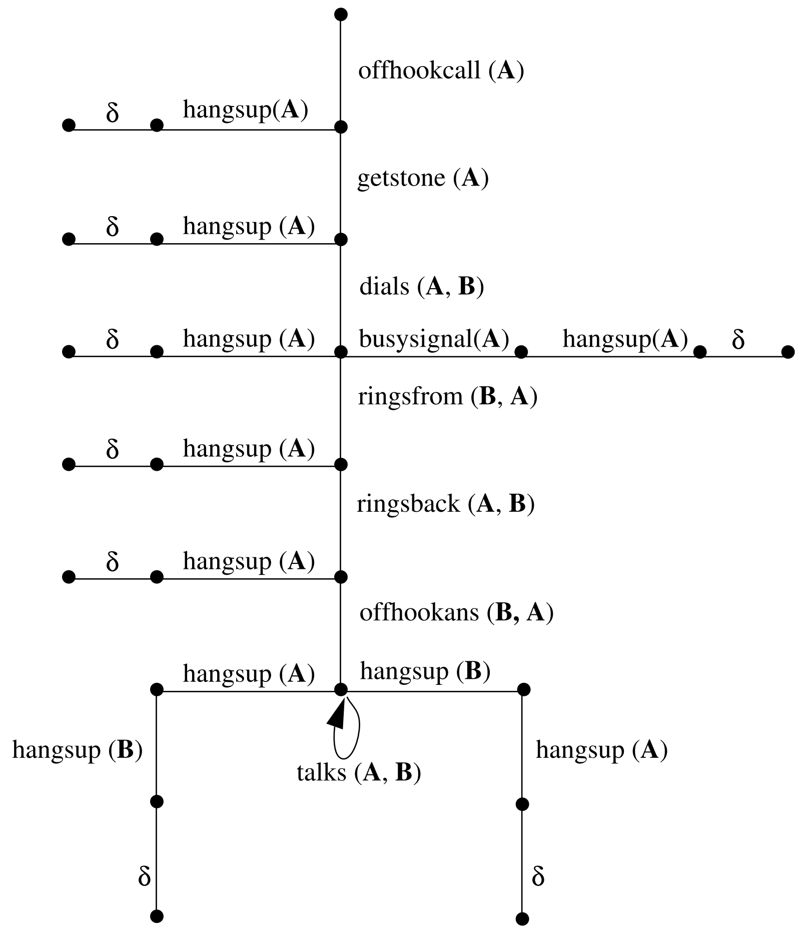


Fig. 4.9 LTS of the controller for a single POTS connection

4.2.4.5 GlobalConstraints

The process *GlobalConstraints* manipulates the data structures of the specification. It synchronizes with *SystemConnections* in order to update the list of busy numbers. The mechanics of updates are best described with respect to the data structure itself, which is the subject of the next section.

4.2.5 *The Data Structures: EngagedSet* and *BusySet*

There is a trade-off between how much information should be exchanged, through processes synchronization, and how much of it should be kept as part of the abstract data types. Choosing the “right” data structure is no simple task. For this specification, our choice is influenced by our objective to focus the reader’s attention on the constraint-oriented style, by keeping the processes simple while exchanging the minimum information required through the service interactions. For example, the only event that requires both the caller number and the called number to be exchanged in the service interaction is the *Dial* event. All other events are associated with either the caller number or the called number. This has the advantage of simplifying the three major processes *Caller, Called,* and *Controller.* But, on the other hand, it complicates the data structure that has to be used. For instance, in order for a caller to receive a *BusySignal*, we must check whether the called number is busy or not. This kind of information is kept inside the process *GlobalConstraints*, by means of a set of pairs.

So, our first data structure *EngagedSet* (or *E* for short) of sort *SetOfPairs* is a set of pairs which has the form: $\{(X, Y) | X \in \{\text{Caller, Called}\}$ and $Y \in \{\text{Called, None}\}$ and *(X, Y)* is different from *(Called, Called)*$\}$. In other words, the first element of a pair in the set is a number which has executed the *Offhook* event. If it is a caller, then the pair (Caller, *None*) is added to the set. If it is a called, then the pair (Called, *None*) is added to the set. The difference between these two pairs is that the first one will be modified, to become (Caller, Called), when the *dials* event is executed, while the second pair remains, in the set, in that form until it is removed as a result of the *Called* executing the *hangsup* event.

If several callers execute the *dials* event while attempting to call the same number, only one of them will succeed to make the *Called* ring. Therefore, we must remember which numbers have executed the *Rings* event so that they may not ring again for another caller. To do so, we defined a second data structure *BusySet* (or *B* for short of type *DecSet*, which is a set of single elements and has the form: $\{X | X \in \{\text{Called}\}\}$. So, if the *Called* is not busy, the *Rings* event is executed and the associated phone number is added to the set *BusySet*. However, if the called number is busy then the caller must receive a busy signal. Again, a called number is busy if it has executed the *Rings* event, as just stated, or if it is the first element of a pair in *EngagedSet.*

Finally, there is the *hangsup* event. When the action containing this event is executed, two cases are distinguished. First, if it is associated with a called number then both the *Rings* event and the *Offhook* (to Answer) event must have been executed. Consequently, the called number must appear both in *EngagedSet* as a pair (Called, *None)* and in the *BusySet* as *Called.* In this case, the called number is removed from both sets. In the second case, when it is associated with a caller number, the algorithm gets a bit more complicated for the following reasons: the caller may hang up

(1) before the *dials* event is executed;

(2) after the *dials* event is executed, but before the *Rings* event is executed;

(3) after the *Rings* event is executed but before the *Offhook* (To Ans) event is executed;

(4) after the *Offhook* (To Ans) event is executed.

For case (1), the called number is not identified yet, and therefore removing the pair (Caller, *None)* from the *EngagedSet* is sufficient. Case (2) is similar to case (1), except that the called is already identified and removing the pair (Caller, Called) from the *EngagedSet* is sufficient as well. The third case means that the *Called* number associated with the *Caller* number which has hung up is already in the *BusySet* and it is not sufficient to remove the pair (*Caller, Called)* from the *EngagedSet*, but we must also remove the *Called* from the *BusySet*. Case (4) is similar to case (3) except that the *Called* number appears three times in the two sets: as (*Caller, Called)* and *(Called, None)* in the *EngagedSet* and as a single element in the *BusySet.* Therefore, when *(Caller, Called)* is removed from the *EngagedSet* and *Called* is removed from *BusySet*, there is still an occurrence of *(Called, None)* in *EngagedSet,* which guarantees that the *Called* is still busy.

4.3 Extending POTS with Features

In this section, we generalize our model so that we are able to use it for the specification not only of basic call processing, but also of telephone features. For the time being, we define a feature as an extension of the functionality of an existing telephone system. In general, a feature extends either the calling side or the called side. To extend a system with a new functionality, we first decide on the *role* of the feature, which can be derived from its informal description. Then, once we understand how the feature is expected to work in the context of the basic service, the integration of the feature's behaviour into the system is accomplished by making the appropriate modifications to the user on which the feature is to be activated, as well as to its controller. Complete details about specifying features are given in [Hibo94]. However, to illustrate these concepts, we present an extension of the POTS specification with the *Three Way Calling (Twc)* feature.

4.3.1 Extending POTS with Twc

4.3.1.1 Informal Description (Role of Twc)

A Twc feature allows a user **A**, on which the feature is active, to establish a connection (i.e., talking session) between two other users, **B** and **C**, simultaneously. First, **A** establishes a connection with **B**. Then, while talking to **B**, **A** sends a special signal, by way of a *flashhook*, to the switch. The switch responds by putting **B** on hold and sending a tone to **A**. **A** can now establish a connection with **C** and then send a second *flashhook* to the switch, which responds by joining all three users in a single talking session; **A** may disconnect **C** from the talking session by sending a third *flashhook*. If any of the users hangs up while in a three way talking state, the connection becomes a two way connection between to the two remaining users. If **A** hangs up while **B** is on hold, the switch sends a ring reminder to **A**, and when answered, a connection is re-established between **A** and **B**.

4.3.1.2 Structure of the Extended Specification

In practice, one would not start on the structure of the specification until some analysis is done and some decisions are reached about the components of the specification and their interfaces. However, we found it more beneficial to first give the structure then present the behaviour. Figure 4.10, which shows the extension of a POTS specification with a *Three Way Calling* feature, uses the following notation, in addition to that given in Section 4.2.2 on page 42. A feature is represented by a small triangle, identified with its name, written next to it; the feature and the user on which it is active are enclosed by a dashed vertical oval.

4.3.1.3 Formal specification of Twc

From its informal description, we are able to deduce that *Twc* has a calling role, meaning that the first action which is exchanged between the feature and its environment is initiated by the feature. Before we give the complete behaviour of the extended specification, let us give the formal specification of *Twc*.

To define the local constraints on this feature, which is shown as process *Twc(**A**)* in Figure 4.10, we analyse the sequence of actions that can be exchanged between the feature and its environment, namely the switching system and the subscriber **A**. Figure 4.11 shows the sequence of actions that the feature executes. It starts with a flash hook signal, then continues in a similar fashion as in POTS, until it reaches a talking state with the third user **C**. After the first

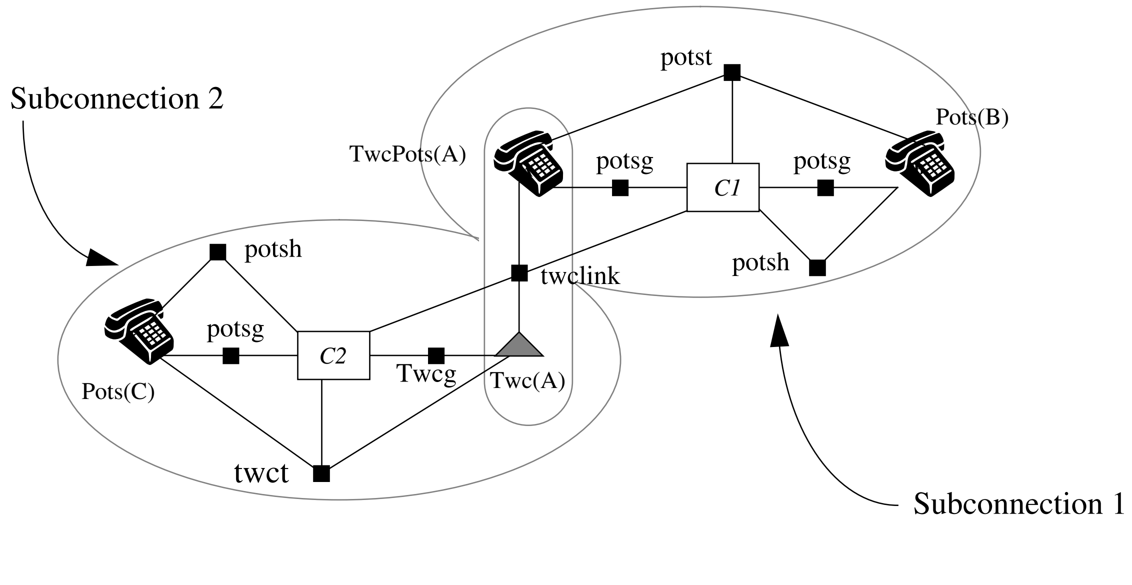


Fig. 4.10 Integration of *Twc* into POTS: top level structure

flashhook, **A** may cancel the invocation of *Twc*, by sending a second *flashhook* before the talking state is reached. If the feature is cancelled, it returns to its initial state, and the next *flashhook* invokes a new instance. If the talking state is reached, the feature allows for a second flash hook, which permits the system to re-activate the connection between **A** and **B** while maintaining **A** and **C** in a talking state. In this example, we assume that **C** has no features, (and discuss below how we represent the fact that B is also connected).

4.3.1.4 The Behaviour of the Extended Specification

The integration of this feature into POTS requires the specifications of the following two sub-connections:

- *Sub-connection 1 := (TwcPots(**A**) |[Potst]| Pots(**B**)) || C1*
- *Sub-connection 2 := (Twc(**A**) |[Twct]| Pots(**C**)) ||* C_2

Note that both Sub-connections have the same general structure that we defined earlier in this Chapter, which shows that our model is general enough to specify connections which include POTS processes as well as feature processes. To complete the specification of Sub-connection 1, we need to modify the calling side of a POTS process (defined as process

Twc (Caller Role)

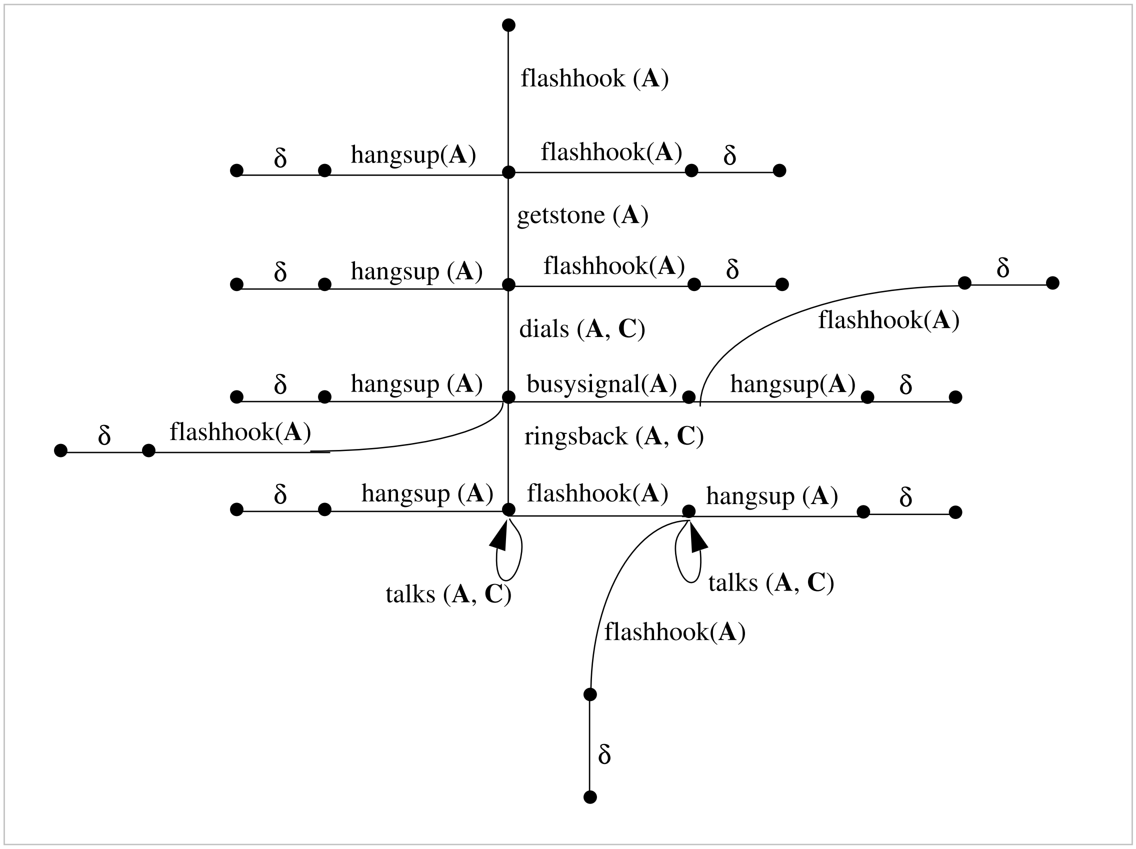


Fig. 4.11 LTS representations of *Twc* (caller role) feature.

*TwcPots(**A**)*) so that it communicates with the process *Twc*(**A**) . We also need to re-specify C_1 so that we can relate *TwcPots(**A**)* and *Pots(**B**)* to each other. To complete the specification of Sub-connection 2, we need the specification of C_2 so that we can relate *Twc(**A**)* and *Pots(**C**)* to each other. Finally, we need to compose the two Sub-connections so that they synchronize on their common actions. Synchronization between the two sub-connections occurs on the gate *twclink*, which is used for exchanging the signals *flashhook* and *hangsup* which occur on **A**.

The design of C_1 expresses the end-to-end constraints of *TwcPots(**A**)* and *Pots(**B**)*. The behaviour of the caller side *TwcPots(**A**)* is given in Figure 4.13; the called side *Pots(**B**)* is a simple POTS process (not shown). Composing these two local constraints defines C_1, as shown in Figure 4.14. Since the actions of *TwcPots(**A**)* and *Pots(**B**)* are included in C_1, it will be sufficient to refer to this process when talking about sub-connection 1.

The design of C_2 expresses the end-to-end constraints of *Twc(**A**)* and *Pots(**C**)*. The caller side of the connection is a *Twc* process whose behaviour is given in Figure 4.11; the called side is a simple POTS process. Composing these two local constraints defines C_2, as shown in Figure 4.15. Since the actions of *Twc(**A**)* and *Pots(**C**)* are expressed by C_2, it will be sufficient to refer to this process when talking about sub-connection 2.

To complete the integration, we compose the two sub-connections as shown in Figure 4.12. They synchronize on the gate *twclink*, which is used to exchange the signals

$$
\begin{aligned}
&(TwcPots(\mathbf{A})\ |[potst]|\ Pots(\mathbf{B}))\ ||\ C1.\\
&|[twclink]|\\
&(Twc(\mathbf{A})\ |[twct]|\ Pots(\mathbf{C}))\ ||\ C_2.
\end{aligned}
$$

Fig. 4.12 Composition of sub-connection 1 and sub-connection 2

TwcPots

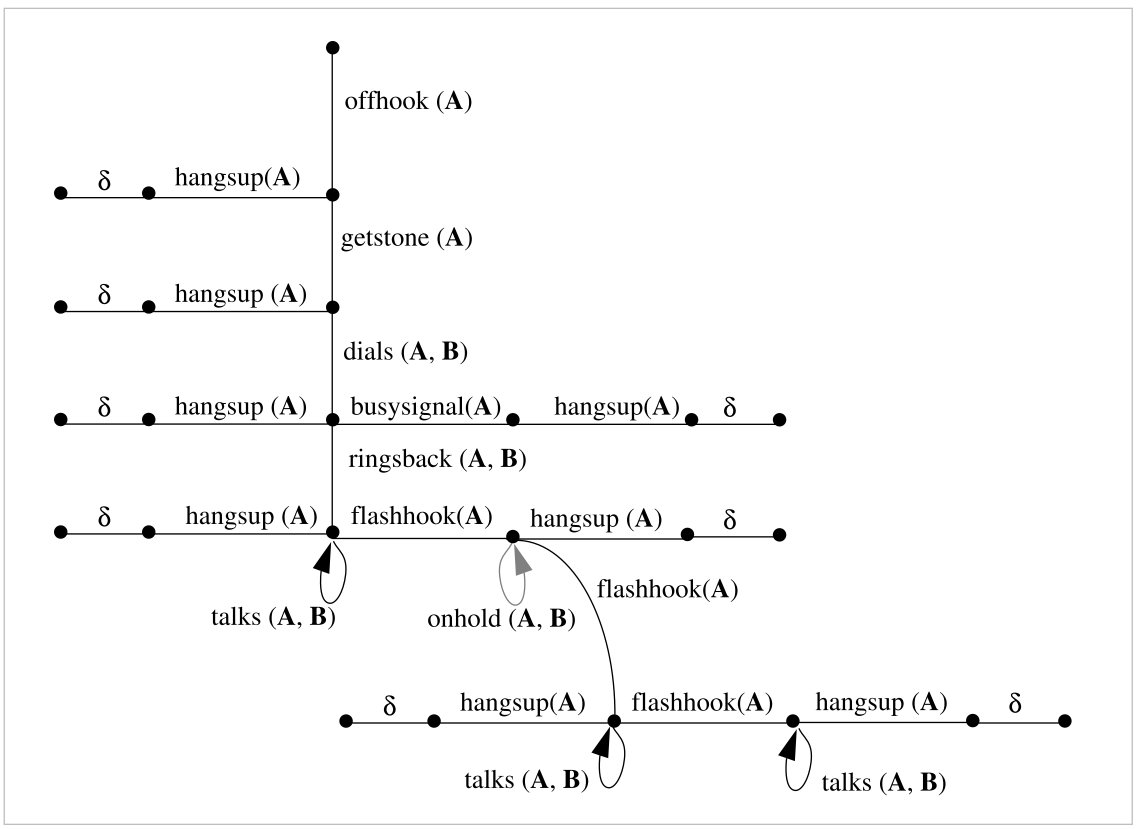


Fig. 4.13 LTS representations of *TwcPots*

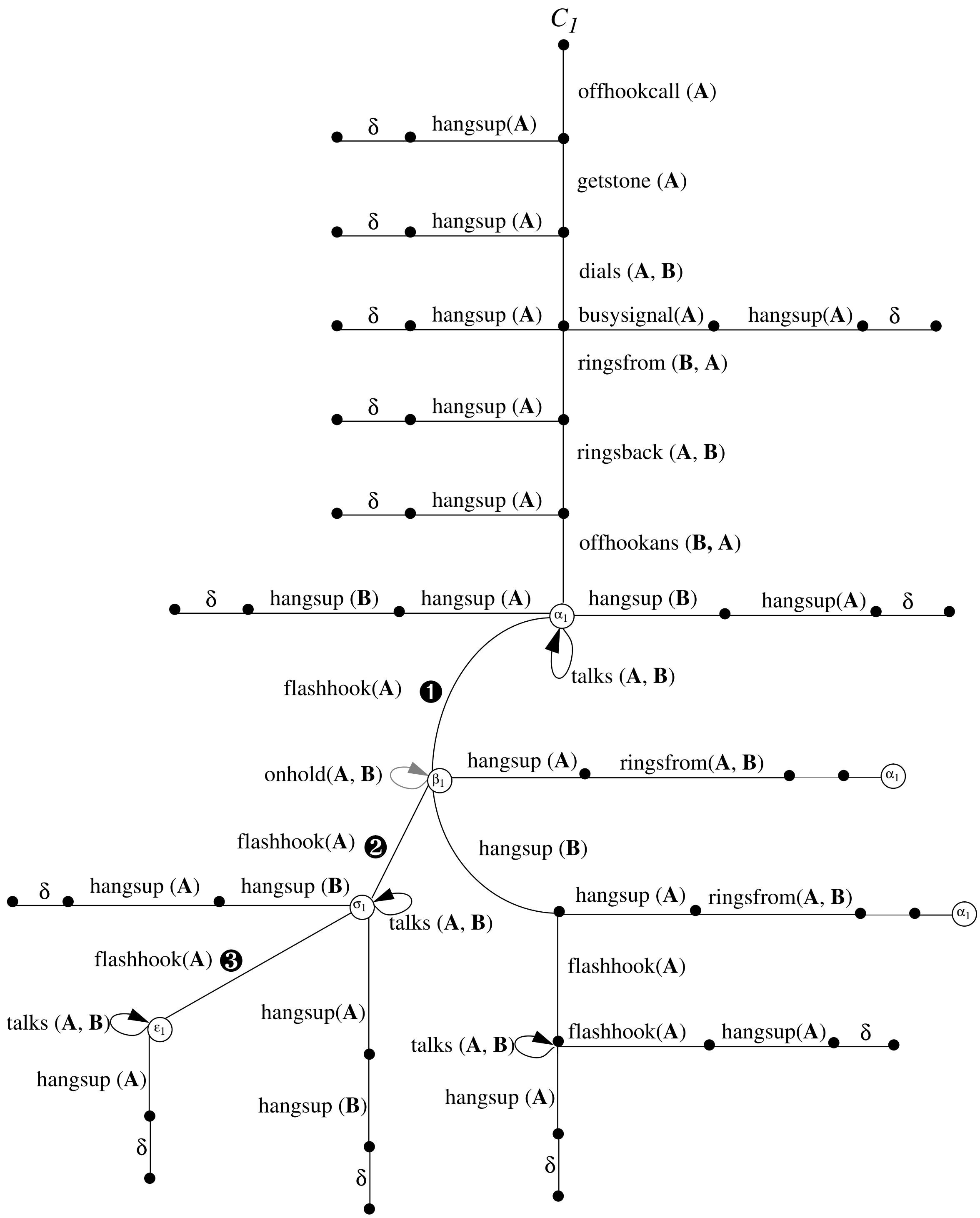


Fig. 4.14 LTS of the controller for sub-connection 1 of a *Twc* feature

flashhook and *hangsup* with respect to **A**. The *flashhook* allows controller C_1, while in the talking state α_1 , to put **B** on hold and transfer control to C_2. This synchronization is identified

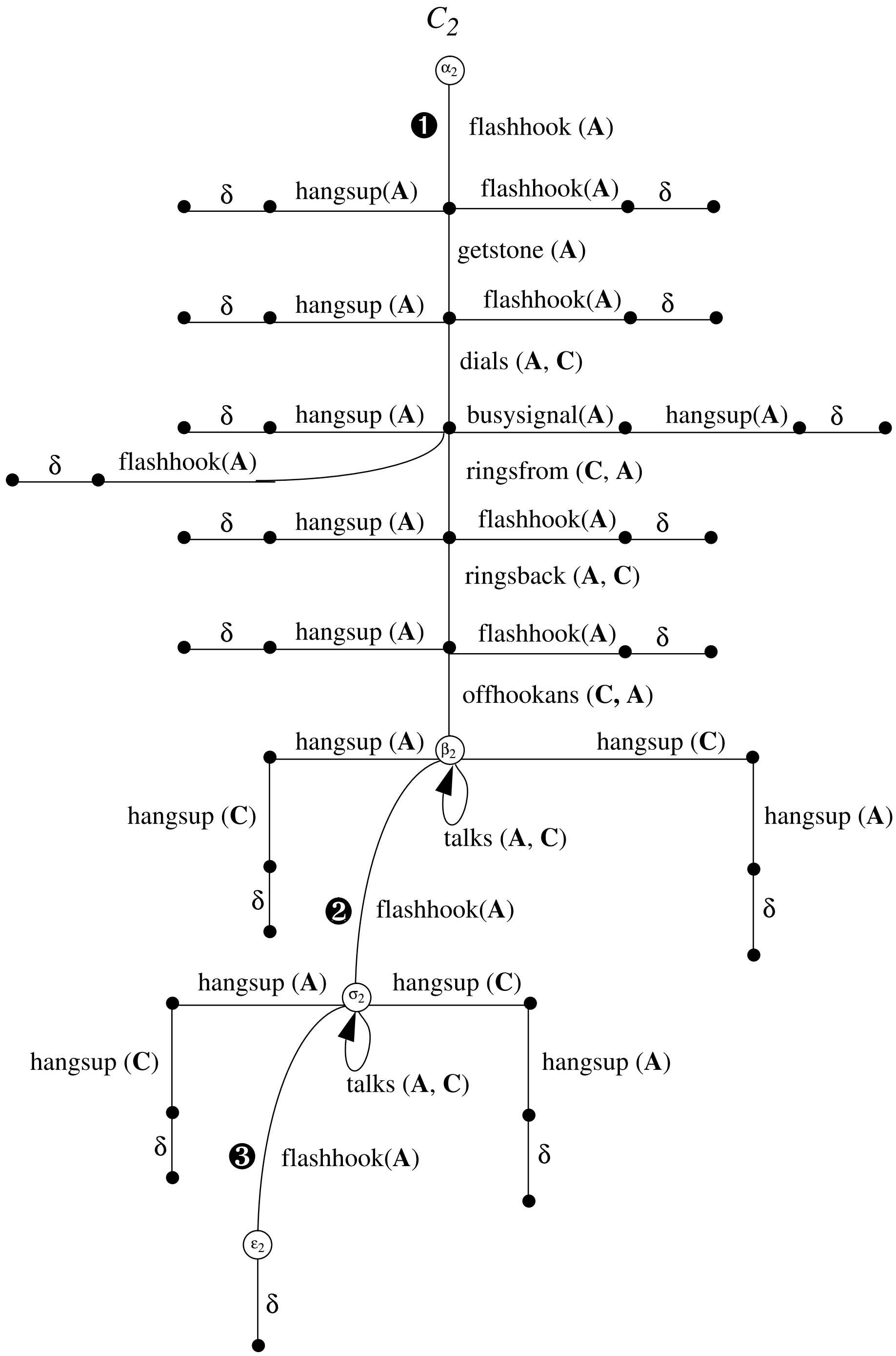


Fig. 4.15 LTS of a controller for sub-connection 2 for a *Twc* feature

by ❶ in Figure 4.14 and Figure 4.15. From this new global state, after the transition, if no hang up occurs, C_1 remains in its local state β_1, while C_2 may continue with its execution until it reaches the talking state β_2. A second *flashhook*, shown as transition ❷, from the state (β_1, β_2)

allows C_1 to bring **A** and **B** back to a talking state while allowing C_2 to maintain **A** and **C** in a talking state as well; this is shown as the global state (σ_1, σ_2). By convention, in our specification, we define this global state as a three way talking state. A more direct way to express this would be to define another gate on which all processes of the two connections (i.e., *TwcPots(**A**)*, *Pots(**B**)*, *C1*, *Twc(**A**)*, *Pots(**C**)*, and C_2) would synchronize. However, this would result in more complex specifications with no additional conceptual value. From this three way talking state, a third *flashhook*, shown as transition ❸, allows C_1 to maintain a talking state between **A** and **B** and allows C_2 to disconnect the communication between **A** and **C**.

Also, note that a hang up from **A** affects both sub-connections, and depending on where the hang up occurs, the two sub-connections exhibit different behaviours. For example, a hang up that occurs from the state (β_1, β_2), where **B** is being held by **A**, allows C_1 to send a ring reminder to **A** in order to re-establish the talking state between **A** and **B**. It also allows C_2 to release the feature.

Finally, each modification of the local or end-to-end constraints requires a modification of the global constraints to support the additional behaviour. In addition to the *BusyList* and *EngagedList* that we described previously, three additional lists are required to maintain the global view of the system, in the presence of *Twc*.

- *TwcList*: A list of single elements, where each element is a *Twc* subscriber; this is a static list. A user can execute a flash hook signal only if he/she is in this list.

- *TwcUsers:* The list of users who have activated the feature. This is a subset of *TwcList.* A user (**A**) is inserted in this list when synchronization occurs of the first flash hook signal. If **A** reaches a talking state with **C**, the second flash hook has no effect on the list. If **A** abandons the call before **C** rings, or if **C** does not answer, the second *flashhook* has the effect of removing **A** from the list. If **C** rings, **A** is removed from the list when the third flash hook occurs, or when **A** hangs up.

- *HoldPairs:* List of pairs (**A**, **B**), where **A** has put **B** on hold, by way of the *flash-hook*. If the second *flashhook* occurs before the talking state is reached, the pair is used to identify the user to which a ring reminder is to be sent. If it occurs after, the pair is removed from the list.

4.3.1.5 Top Level view of the Extended System

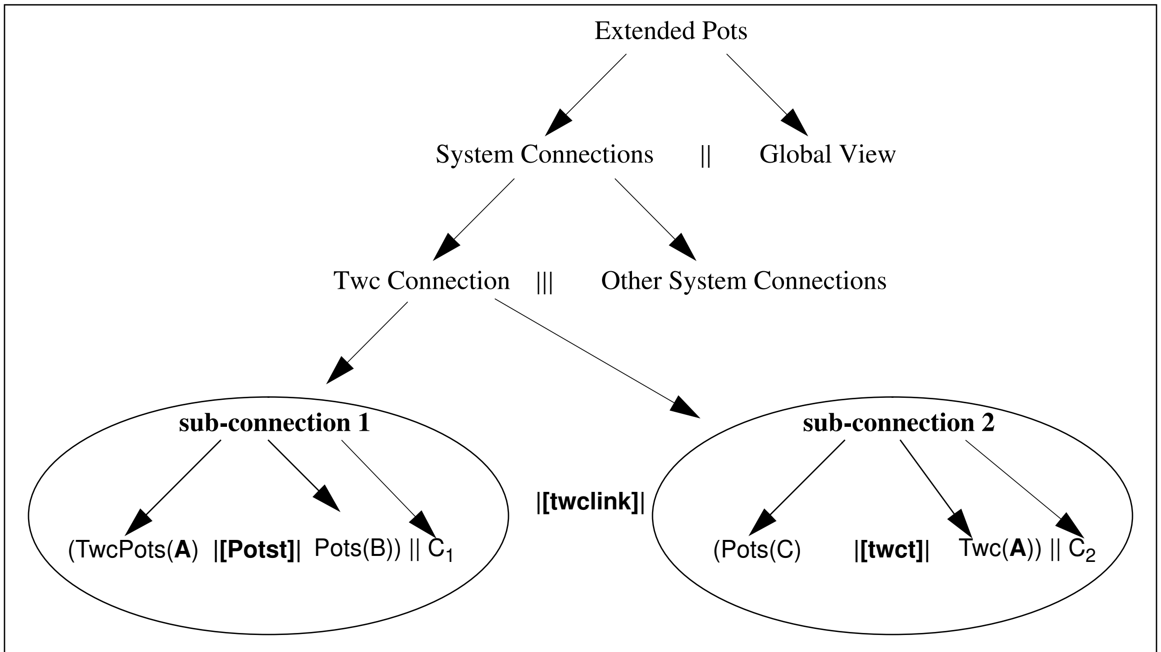


Fig. 4.16 Top level view of extending POTS with *Twc*

The resulting top level specification is shown in Figure 4.16. By following the structure's hierarchy, we see that the system connections now consist of an additional connection, called *TwcConnection*. This connection, which interleaves with the rest of the connections in the system, is represented by the two sub-connections 1 and 2, whose controllers are *C1* and *C2*, respectively; they synchronize on the gate *twclink*.

4.4 Validating the Specification

LOTOS is based on formal semantics, by which specifications could be (in principle) proven correct according to certain criteria. Unfortunately, however, the proof techniques available today are of limited power and do not allow verification of specifications of this size. Debugging is the other option.

LOTOS specifications, if written in an executable style, can be simulated by means of an interpreter [GuHL88]. This allows debugging the specification to increase the designer's confidence that the specification reflects the requirements. The University of Ottawa LOTOS interpreter allows the designer to execute a specification in two ways: step-by-step and composing test processes in parallel with the specification.

4.4.1 Step-by-step Execution

In step-by-step execution mode the user can debug a specification by executing it one action at a time. At each step, the interpreter presents the user with the list of all possible next actions. Users are responsible for choosing the next action and providing appropriate data values. When using this execution mode, one proceeds in a bottom-up fashion. In other words, each leaf process is debugged separately, their parent processes are then debugged, and so on until reaching the root, which is the specification itself. An execution sample of the specification is shown in Figure 4.17. Exhaustive debugging was of course impossible, however many important paths were tested. Note that test sequences are chosen according to our intuitive understanding of how the specification is expected to behave.

As the use of formal methods for specifying systems increases, interest in methods for generation and selection of test cases from the formal specifications becomes important. For more details about the testing domain, we refer the reader to [Brin88], [BALL90], [ISO91], [BrTV91]. The subject of how effective these methods are for detecting feature interactions is left for further research.

4.4.2 Test Processes

Once the most obvious errors are removed, a more efficient way to detect errors in a specification is to compose a non-branching test process in parallel with it, and then obtain the execution tree of the resulting specification [GuHL88]. If the execution reaches the last action in the test process, then the specification accepts the sequence of the test process.

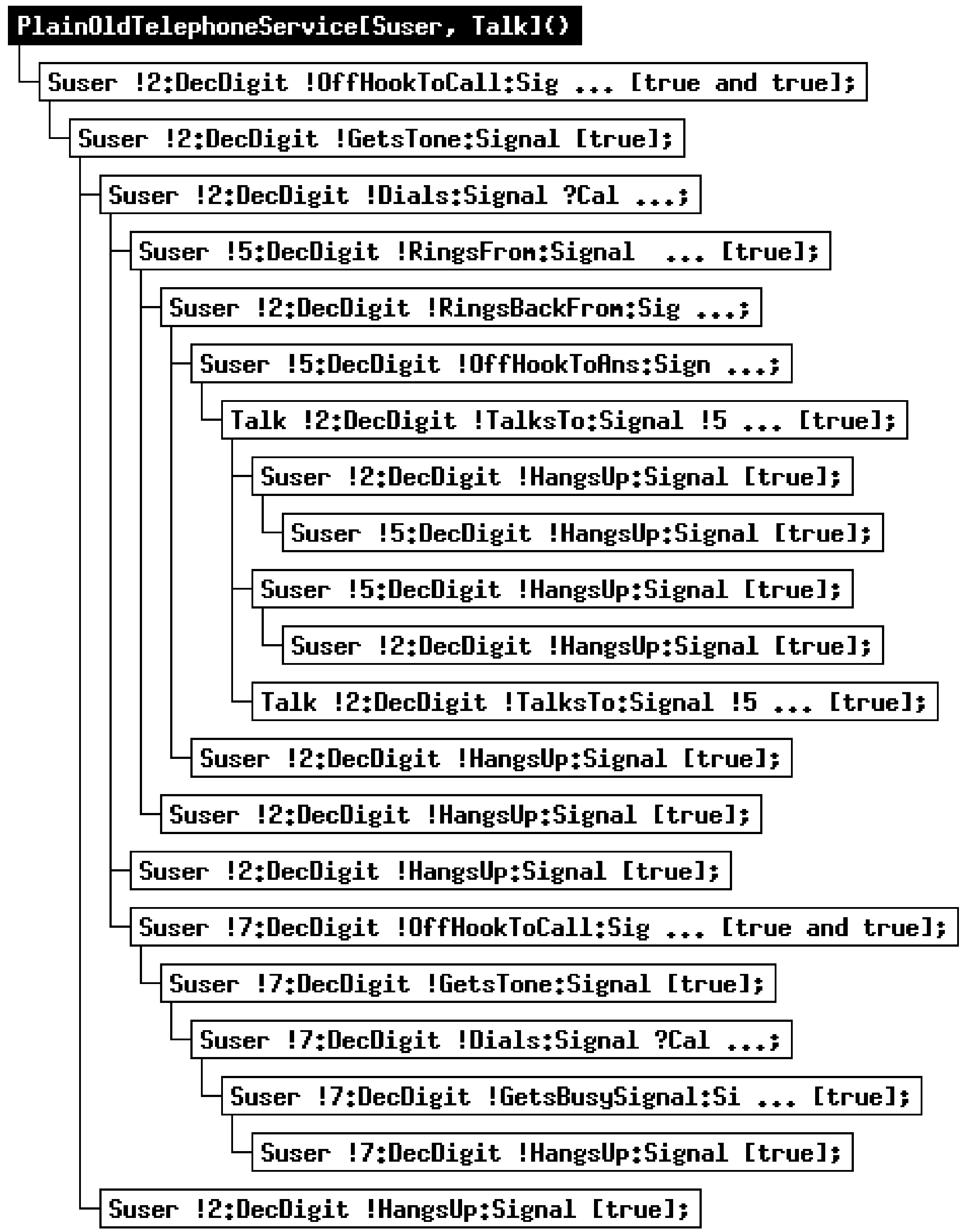


Fig. 4.17 Generation of sample traces of POTS

CHAPTER 5 A Methodology for Detecting Feature Interactions

In this chapter we introduce a methodology for detecting feature interactions. Section 1 gives a general discussion which serves as the basic motivation for our methodology. In section 2, we give the details of the methodology itself, using a hypothetical example for illustration purposes. Finally, in section 3, we explore the application of the methodology using a concrete telephone example.

5.1 Motivation and Background

Figure 5.1 shows our view for integrating new features into an existing system. New features are normally motivated by user requirements, which are communicated to the telephone system designers in the form of imprecise and fragmented suggestions. These are shown as R_1 for f_1 and R_2 for f_2 in the figure. These requirements are then converted into informal descriptions, using natural languages, diagrams, charts, and other visual notations. The informal descriptions become the basis for writing formal specifications which integrate the new features into the existing specification of the system. Specifiers use their knowledge and experience to modify the existing system so that it incorporates each of the new features independently. Note that we emphasize the notion of *separate integration* of features. This is driven by the pragmatic nature of user requirements and their views of the system. Users do not normally get together and decide on all new features to be added to the system. Rather, each requirement for a feature is defined independently of others, very often by different users. These are shown as $Sys[f_i]$, which means that the feature f_i is integrated in the context of the existing system *Sys*. A formal definition of this notion is given later in Definition 5.1 on page 68. The next step is to merge the behaviour of all independent integrations $Sys[f_i]$ into a single specification. This is shown as ***Formal Specification 1*** in the figure. This specification becomes

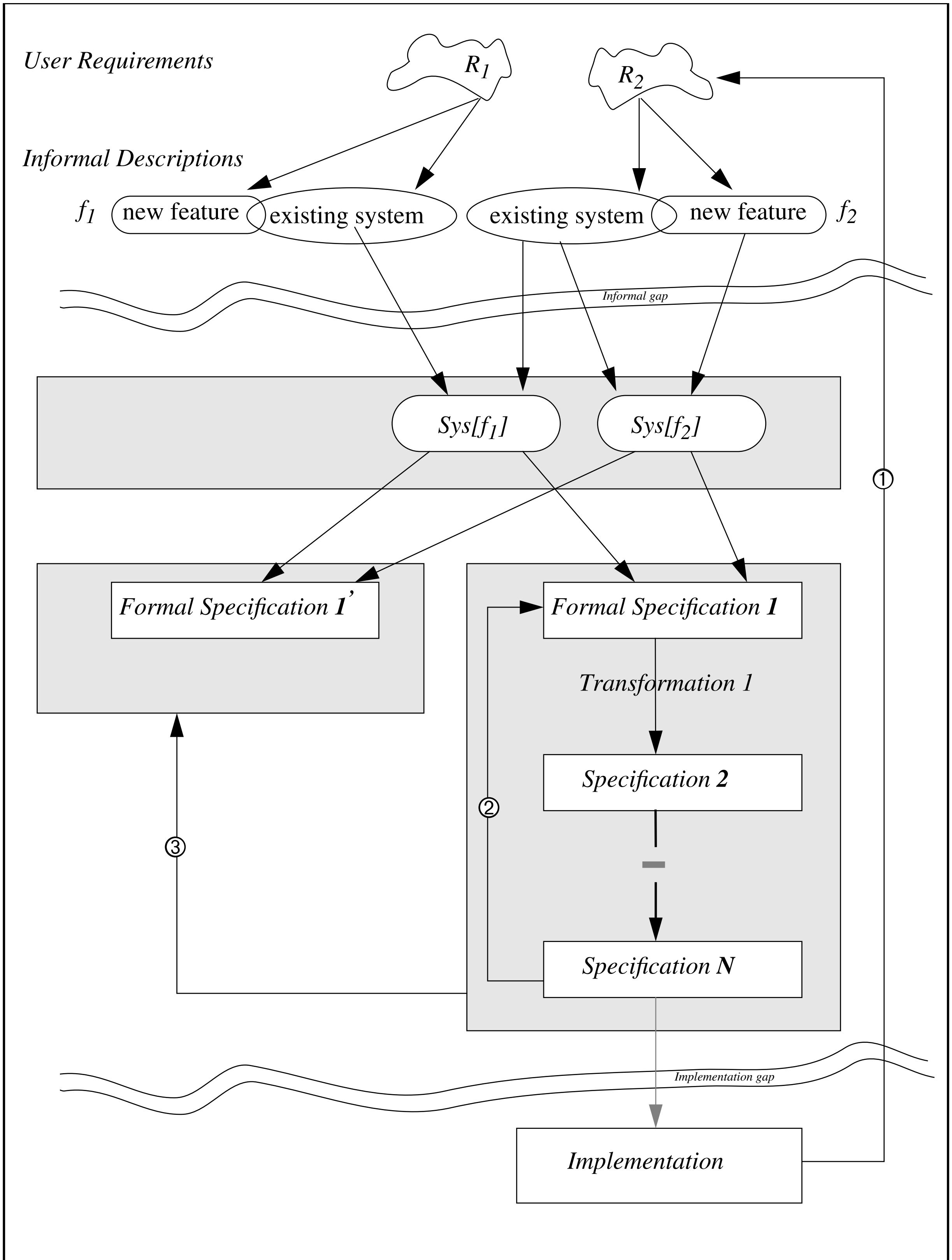


Fig. 5.1 Life cycle for integrating new features into a system

a reference point on which a fully operational *implementation* is to be based. The specification undergoes a series of transformations, the last of which is the *implementation*. This last stage raises several interesting questions, one of which is the problem of *conformance*.

The conformance problem[ISO91] can be stated in the following way. Given a set of user requirements, does the implementation actually perform the desired functionality that was expressed by the requirements? Establishing a formal conformance relation between these two levels of the system, as indicated by line ① in the figure, may be impossible for two reasons. The first one is the *informal gap* that separates the informal descriptions from the formal specifications. The second reason is the *implementation gap* that separates the last stage of the refined formal specification ***N*** from the implementation phase.

However, it is still quite useful to consider a conformance relation between the two levels of **specification *1*** and **specification *N***, as indicated by line ②, for the following reasons:

- **Specification *1*** can be seen as a reference point by a software developer to produce an implementation of the system, which is valid only if it conforms to its specification;
- **Specification *N*** can be seen as a refinement of **specification *1*** and should, therefore, be as close as possible to an implementation. Furthermore, if **Specification *N*** is derived from **Specification *1*** according to some formal relations, then we would expect that **Specification *N*** conforms (in some precise meaning) to **Specification *1***;
- An interesting body of work has been developed which allows us to reason about conformance relations using a single formalism, the labelled transition system formalism.

What remains to be explained in Figure 5.1 are the relations between the three shaded areas and how to use the conformance relation to relate **Specification *1*** (or any of the conforming specifications from ***1*** to ***N***) to **Specification *1*'**, as shown by ③. This is the starting point for the development of our methodology which we present in section 5.3, but first some mathematical concepts and notations are in order.

5.2 A Methodology for Detecting Feature Interactions

In this section we propose a methodology for detecting feature interactions. We start by giving the steps of the methodology in point form. This will help the reader focus on the global picture while reading the details of each step. In fact, we recommend that Figure 5.2 be used as a reference point while proceeding with the rest of the section.

There are two important assumptions that we are making about the methodology. They will be restated in their appropriate context, but it is worth highlighting them at this point. First, we assume that, *from a specification point of view,* features do not "build" on each other, meaning that when a feature is integrated into the system, no assumptions are made about the behaviours of other features in the system. Second, features are defined independently of each

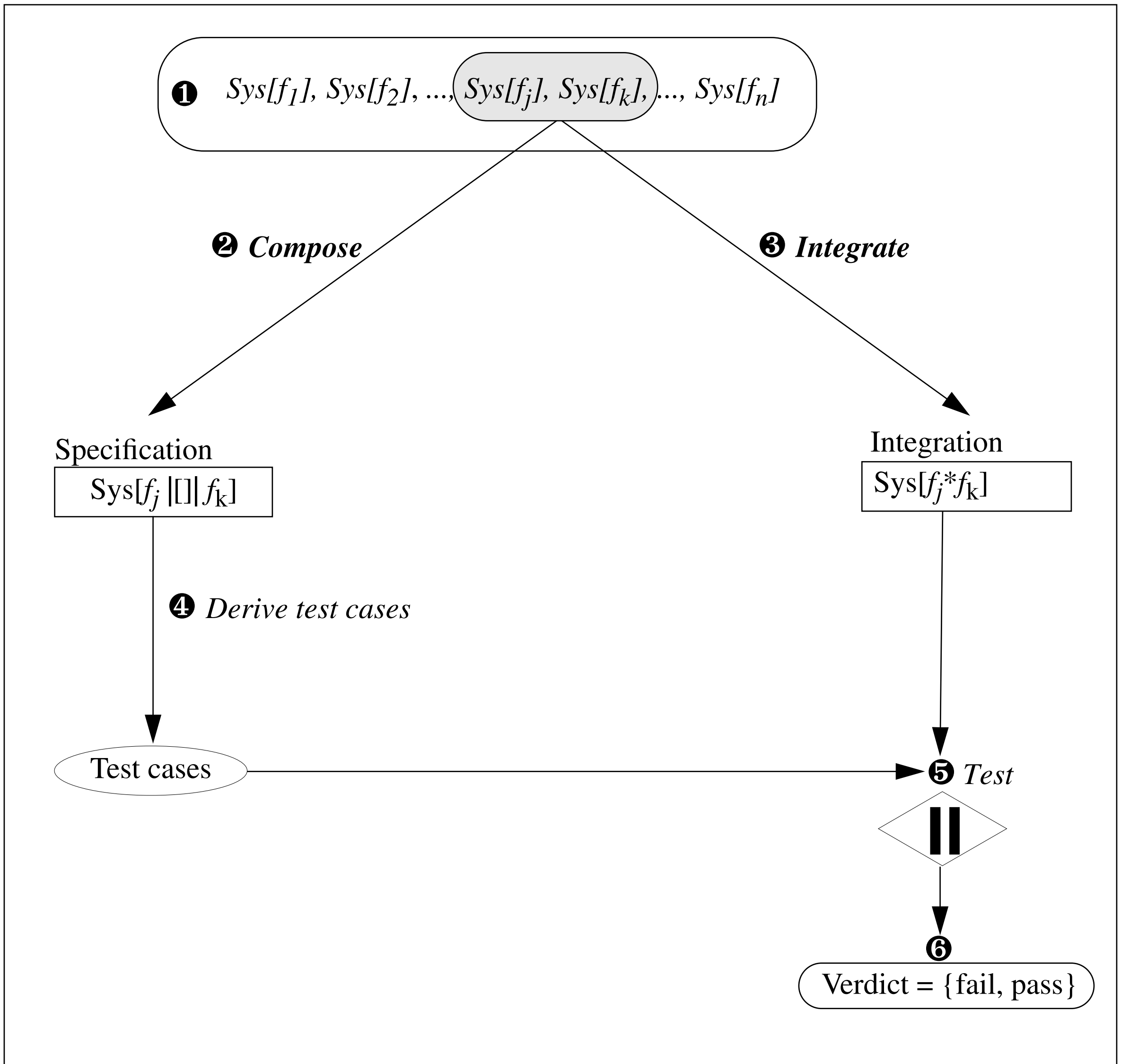


Fig. 5.2 Methodology for detecting feature interactions

other. This implies that features have independent alphabets at the specification level, although some of their alphabets may be mapped onto a common event in the integration. If such a mapping occurs, ambiguities may be created as will be illustrated later in the case of call waiting and three way calling.

Each step will be explored in detail in the remainder of this section. A brief statement about each step follows:

❶ *Specify each feature independently, within the context of the existing system, using the notion of constraints.*

❷ *Compose the n features into a single specification so that they are able to synchronize on their common interaction points (with respect to the system) and interleave on the rest of their actions. Consider the result of this composition as a specification with respect to the integration obtained in* ❸ *below.*

❸ *Integrate the n features into a single specification (which includes the n features and the existing system) so that each feature is able to perform its function when the other features are disabled. We will refer to this as the integration of the n features.*

❹ *Derive a set of test cases, using the theory of the derivation of tests for LOTOS processes, from the specification obtained in* ❷ *above.*

❺ *Simulate the system obtained in step* ❸ *using the test cases obtained in step* ❹*, and check for deadlocks.*

❻ *Interpret the results in the following way. A deadlock in* ❺ *implies that the way these two features are integrated in the system does not allow for their simultaneous activation.*

5.2.1 **Specification of Features in the Context of a System** (step ❶)

Figure 5.3 (a) shows the POTS model that we defined in chapter 4 for the specification of basic call processing. In this section, we show how to extend the model, as given in Figure 5.3 (b), so that it can be used for the specification of telephone features as well. To extend a system with an additional behaviour, we first decide on the *role* of the feature. In general, each feature can be classified as acting either on behalf of the caller process or the called process. Once that decision is made, the integration of the feature's behaviour into the system is accomplished by integrating the feature, using local constraints, into the process on which the feature is to be activated. This results in a new process which requires a modification

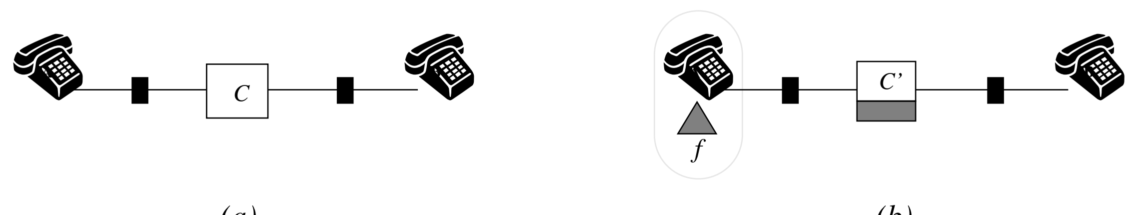


Fig. 5.3 Extending the POTS model to support features

of the end-to-end constraints expressed by the controller of POTS. In the above figure, C' is obtained by modifying C in order to support the functionality of the new feature. For each feature, the specifier must decide how to extend the POTS system so that it supports the new feature. We assume that this step is carried out manually. Its success and efficiency depend on the knowledge and experience of the specifier. The resulting specification of integrating f_i into POTS is called the behaviour of f_i in the *context* of POTS. Formally,

Definition 5.1: System Context

We define $Sys[f_i]$ to be a behaviour where the following condition holds:

$\forall\ t \in Trace(f_i), \exists\ t' \in Trace(Sys[f_i])$ such that $t \sqsubseteq t'$.

As an example, suppose that f_1 is given by the tree of Figure 5.4 (a) and the result of its integration into POTS is given by the tree of Figure 5.4 (d), then we have:

- $Trace(f_1) = \{\langle\rangle, \langle x\rangle, \langle x\delta\rangle\} = \{t_1, t_2, t_3\}$
- $Trace(Pots[f_1]) = \{\langle\rangle, \langle\alpha\rangle, \langle\alpha\beta\rangle, \langle\alpha x\rangle, \langle\alpha\beta\delta\rangle, \langle\alpha x\delta\rangle\} = \{t_1', t_2', t_3', t_4', t_5', t_6'\}$;

Since $t_1 \sqsubseteq t_1' \in Trace(Pots[f_1])$, and

$t_2 \sqsubseteq t_4' \in Trace(Pots[f_1])$, and

$t_3 \sqsubseteq t_6' \in Trace(Pots[f_1])$

Then we say that f_1 is specified in the context POTS. Using similar deductions we can show that f_2 is also specified in the context of POTS, as shown in Figure 5.4 (c) and (e).

5.2.2 Composition Vs. Integration of Features (steps ❷ and ❸)

Our primary objective, as we have already mentioned, is to answer the following question: is there an interaction between features f_1 and f_2 when they are integrated into an existing system? To answer this question, one must define a reference point, which we call

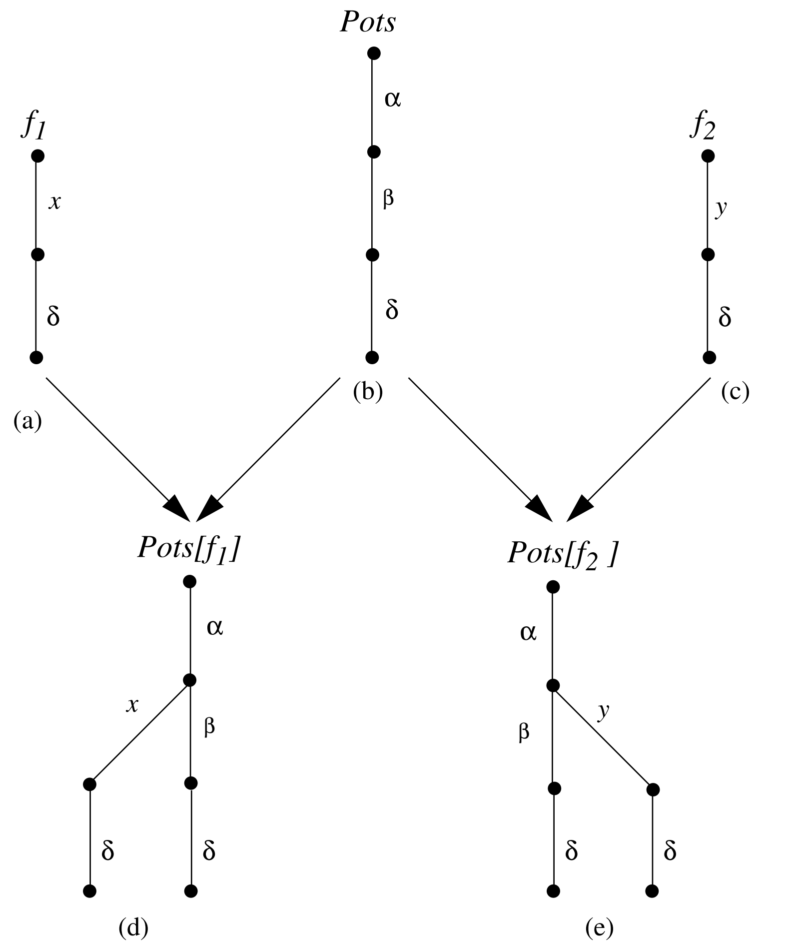


Fig. 5.4 Integrating a feature into a system.

composition, against which an answer can be evaluated. We base our definition on the notion of *simultaneous* execution. For practical purposes, this notion is interpreted in the context of interleaved semantics. Saying that two features can execute *simultaneously* is equivalent to saying that each feature will reach its terminal states and that, from any given state along the execution path, their actions are allowed to execute in any order. In practice, however, when specifiers produce specifications which integrate the functionalities of several features, their primary concern is to include the functionality of each feature, one at a time, in the resulting specification. We will refer to this as an *integration*. For each feature that is being integrated, the specifier gives little or no consideration to what effects this will have on other features in the system. Figure 5.5 puts these two notions in perspective. Given two features f_1 and f_2, parts (a) and (b) show the integration of each of the two features in the context of POTS. Part (c) shows their compositions and part (d) shows their integration, assuming that *x* and *y* are possible after α only.

In this thesis, we assume that the integration is done by hand, where the features are added one at a time by analysing the POTS system, into which the features are to be integrated, and for every state of POTS, we would decide whether or not actions from the feature under consideration are allowed. An underlying formal model for merging for specifications, which would correspond to adding features to POTS in our case, is the subject of active research [EKDB92].

The composition of features turns out to be conveniently expressed using the LOTOS composition operator "$|[a_{pots}]|$", where a_{pots} are all the actions that are common to both $Pots[f_1]$ and $Pots[f_2]$, the independent integrations of f_1 and f_2 in the context of POTS. For conciseness, we will refer to the *composition* by $(Pots[f1\ |[]|\ f2])$ and the *integration* by $Pots[f_1{*}f_2]$. Note that this notation is easily extended to *n* features.

On the other hand, the requirements imposed on the system designer to integrate the two features into the system, are as follows:

- $Trace(Pots[f_1]) \subseteq Trace(Pots[f_1{*}f_2])$ (r1)
- $Trace(Pots[f_2]) \subseteq Trace(Pots[f_1{*}f_2])$ (r2)

meaning that the functionalities of both features, when considering the behaviour of one feature at a time, are preserved in the integration of f_1 and f_2 in the context of POTS. However, requirements r_1 and r_2 are not sufficient to guarantee that the execution of the LTS of Figure 5.5 (d) always reflects *a desired* joint behaviour. In other words, the fact that the traces of f_1 in the context of POTS are a subset of the traces of the integration of f_1 and f_2 in the context of POTS, *is not a sufficient condition* to conclude that the traces of f_1 always execute to their completion, even when interleaved with the traces of f_2, because it may be the case that once an action from a trace of f_2 is executed in the middle of a trace from f_1, the system moves to a global state from which the next action of f_1 is no longer possible. Several examples of this are explored in Chapter 6.

Logical design errors must be discovered and removed at the specification level. In our case this means that the LTS of Figure 5.5 (d) should be analyzed further to make sure that the

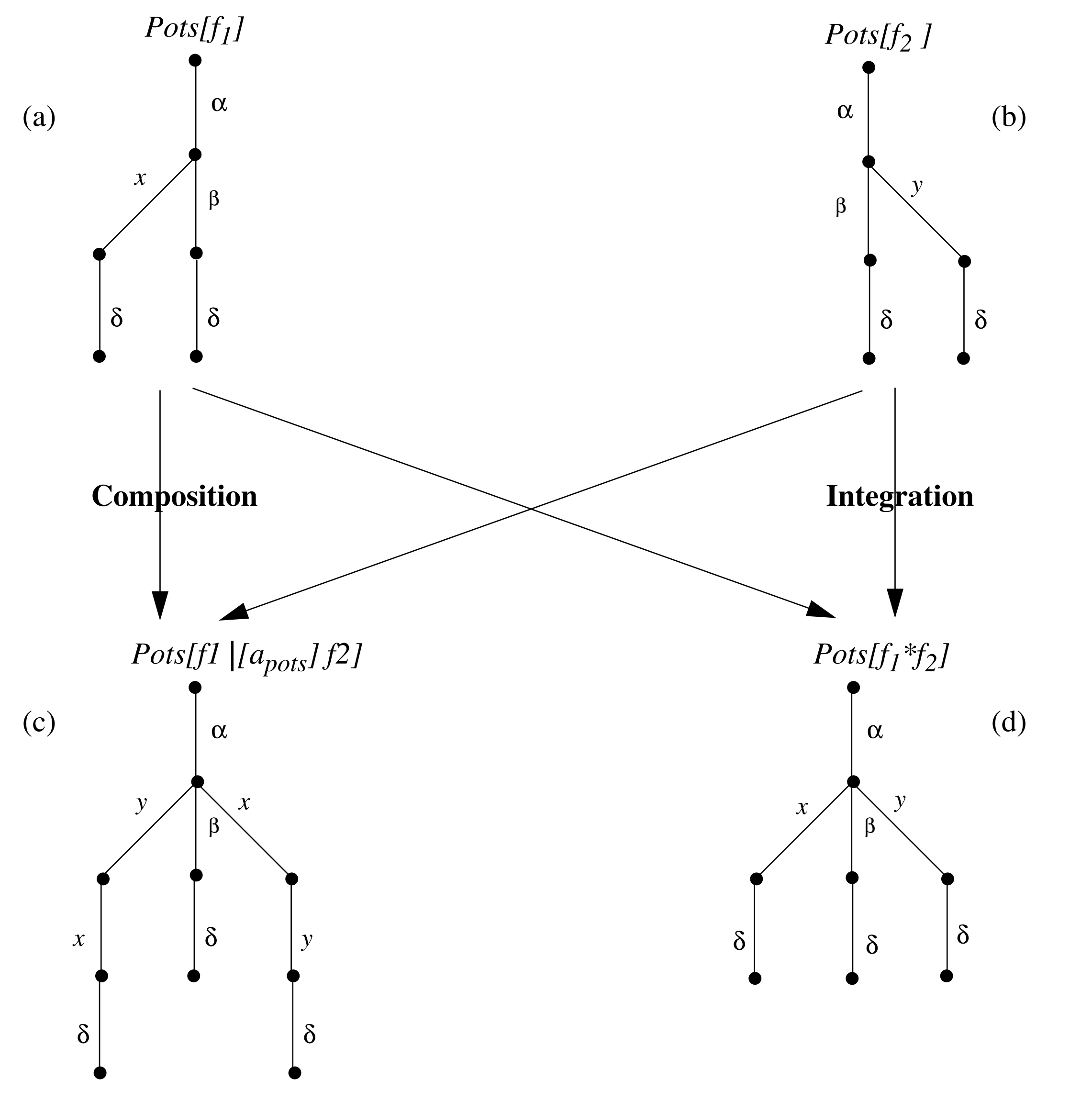


Fig. 5.5 Integration Vs. Composition of Features

two features do not exclude each other during the execution. Potential interactions between the features must be discovered and included as part of the final specification, on which the implementation is to be based.

5.2.3 Derivation of Test Cases to Detect Interactions (step ❹)

5.2.3.1 Conformance Testing and the Detection of Feature Interactions

In this section we review the results reported in [BrSS87, Brin88, BrAL90] and show

how the notion of *conformance testing* has a direct application in the domain of detecting feature interactions. But first, let us define some concepts which are needed for conformance testing.

Definition 5.2: Deadlocks in Terms of Refusals

Let L be the set of observable actions, L^* be the set of traces. Then, for I a labelled transition system, $A \subseteq L$, $\sigma \in L^*$:

- ***I Refuses*** (σ, A) is defined as: $\exists I' : (I = \sigma \Rightarrow I'$ **and** $\forall a \in A: I' \neq a \Rightarrow)$
- ***I Deadlocks***(σ) is defined as: ***I Refuses***(σ, L)

The first part of the definition says that *I* may execute the trace σ, and after doing so, refuse every action in the set A. Similarly, the second part of the definition says that *I* reaches a deadlock if it refuses every observable event, after executing σ.

A detailed treatment of the conformance testing problem is given in [BrSS87] [Brin88], in which a formal framework is defined. Conformance allows one to reason about two specifications using a single formalism. In this context, the first specification is taken to be an abstract representation of a physical realization and the second specification a description of the system's desired behaviour. Therefore, to check whether *the integration (I) of two features conform to their composition (C)* corresponds the following definition [Brin88].

Definition 5.3: Conformance

Let C and I be processes. We say that :

I **conf** C $\Leftrightarrow \forall \sigma \in$ Trace(C), $\forall A \subseteq L$,

if I Refuses (σ, A) **then** C Refuses (σ, A).

Informally, *I* conforms to *C* if, and only if, testing the integration against the traces of *C* does not lead to deadlocks that would not occur while testing the composition against those same tests. In other words, testing the integration does not produce deadlocks that would not be discovered while testing the composition.

We are now in a position to refer back to Figure 5.1 on page 64, to explain the relation between the two shaded areas connected by line ③. With respect to the area on the right, we assumed that **specification *N*** is an abstract representation of the system's realization. We can further assume that **specification *N*** conforms to **specification *1***, according to the conformance relation. However, the conformance relation does not guarantee that the two features can execute their actions, excluding those in the synchronization set with POTS, in an *interleaving manner*. This interleaving is expressed by **specification *1*'** in the shaded area on the left. This is the relation that we wish to verify. It leads us directly to the formalization of the feature interaction problem. To understand the following definition, recall that we are interested in determining whether all the traces of each feature can execute to completion in the integration Sys[f_1*f_2*...*f_n] of the features. By [GaLO91], a system that has all such traces is given by $Sys[f_1 \,|[]|\, f_2 \,|[]|\, ... \,|[]|\, f_n]$.

Definition 5.4: Formalization of the Feature Interaction Problem: Int

Let $f_1, f_2, ..., f_n$ be features,

Let A be the alphabet of POTS and A_i the alphabet of f_i such that:

$\forall\ i, j{:}\ A_i \cap A_j = \emptyset$ and $A \cap A_i \neq \emptyset$, for $1 \leq i \leq n$, $1 \leq j \leq n$.

Let C and I be processes, such that:

$C := Sys[f_1 \,|[]|\, f_2 \,|[]|\, ... \,|[]|\, f_n]$ and $I := Sys[f_1 * f_2 * ... * f_n]$,

Then, we say that : $\mathbf{int}(f_1, f_2, ..., f_n) \Leftrightarrow \neg(\text{I } \mathbf{conf} \text{ C})$

This means that an interaction exists between the n features if, and only if, the *integration* of the features ***does not*** conform to their *composition*. In other words, if the integration does not conform to the composition, which includes the traces of all the features with respect to POTS, then by the conformance definition, there must exist an action (of a given trace) which is valid with respect to the composition, but not with respect to the integration. This implies that one (or more) of the features in the integration has modified the behaviour of another feature, by preventing it from executing one of its valid traces, as expressed by the composition.

In the definition above, we assume that $A_i \cap A_j = \emptyset$. This is consistent with our intuitive views of the features. At the abstract level, features are defined in terms of their interactions with respect to POTS only. However, it is possible that two actions of two different features may be mapped onto the same element in the integration, such as the *flashhook* signal which is defined both in the context of call waiting and three way calling. So a problem such as this one, which we can also detect, is not due to the definition of the alphabets of the features, it is due to the mapping of the actions.

Definition 5.4 is the crucial link that allows us to exploit the results reported in [BrSS87], [Brin88, [BrAL90]. For completeness purposes, we dedicate the remainder of this section to an overview of those results. For more details, the interested reader is referred to the cited literature.

5.2.3.2 Derivation of Tests

In this section we give a brief overview of the theory of test derivation from a LOTOS specification, based primarily on the results reported in [Brin88, PiFr90]. The theory assumes that processes, which are expressed in terms of LOTOS behaviour expressions, have only successful terminations, meaning that whenever a process reaches a deadlock, it must have done so via the successful termination **exit**. The theory also asserts that there exists a *canonical* tester T_c that can discriminate those integrations which conform to the composition and those which do not. For an integration I and a canonical tester T_c, this discrimination is accomplished according to the following **passes** relation.

Definition 5.5: passes relation

Let I and T_c be processes, then

I **passes** T_c iff $\forall\ \sigma \in L^*$: *If* $(I \,||\, T_c)$ after σ **deadlocks**

Then T_c after σ **deadlocks**

For a composition C, a canonical tester T_c is defined as follows:

- *Traces* (T_c) = *Traces* (C), and
- $\forall$ I: I **conf** C iff I **passes** T_c

In addition, each T_c can be expressed as a set of testing processes called the *irreducible reductions* (IR$_s$) of T_c. Formally,

IR$_s$ $=_{def}$ $\{T |\ T$ **red** T_c, $\forall\ T'$: T' **red** $T \Rightarrow T' = T\}$, where

B1 **red** B2 iff B1 **conf** B2 **and** *Traces*(B1) $\subseteq$ *Traces*(B2)

Figure 5.6 captures the intuition behind these definitions.

The integration of two features into a system is driven by the functionalities of both the features and the existing system. Since it is not possible to define the semantic of a general model for the integration of any two given features, we give an example to show that, depending on the results of the integration, an interaction may or may not occur. In either case, our

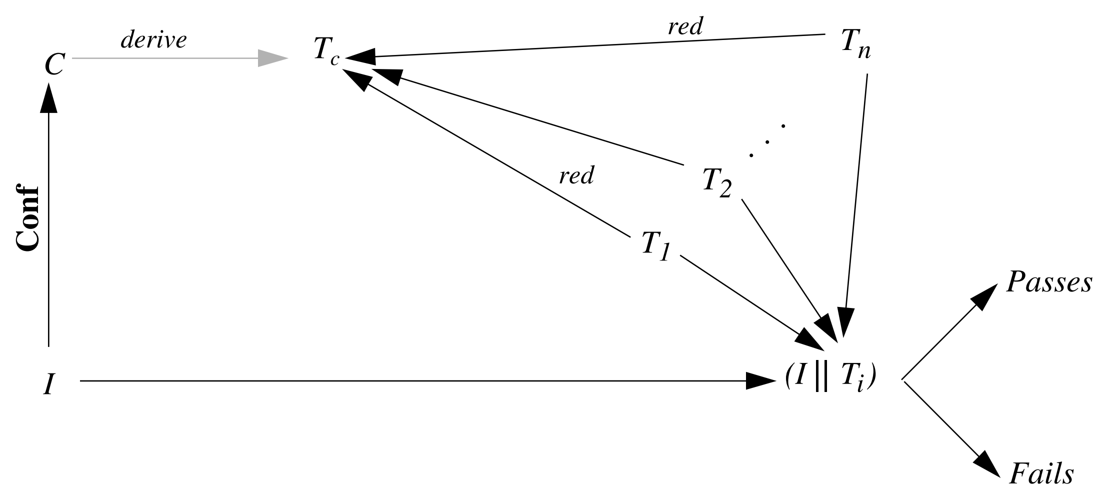


Fig. 5.6 Relation between *composition, integration* and conformance

methodology succeeds in reaching the correct verdict. It is important, however, to keep in mind that the ***passes*** verdict simply means that more testing is required, whereas a ***fails*** verdict means that an interaction is detected.

To illustrate the steps that we explored so far, let Sys := α; β; **exit** be an existing system to which we wish to integrate two features f_1 := x; **exit** and f_2 := y; **exit**. Also, let us assume that the desired global behaviour of the system requires that x executes after α, with respect to f_1, and y executes after α, with respect to f_2. Then, the integration of these two features is expressed by I := Sys$[f_1 * f_2]$ = α; (x; **exit** [] β; **exit** [] y; **exit**), as shown in Figure 5.7. Notice that a choice between action *x* and action *y* is introduced as a result of integrating both features into *Sys*. This is consistent with the desired behaviour.

According to our definition **int** of feature interaction, an interaction exists between f_1 and f_2 if, and only if, ***I does not conform*** to *C*, where

C: = Sys[f_1 |*[]*| f_2]

= α; (β; **exit** [] x; **exit**) |[α, β]| α; (β; **exit** [] y; **exit**)

= α; (x; y; **exit** [] β; **exit** [] y; x; **exit**)

Then, the canonical testing process T_c can be expressed by the tree of Figure 5.8. In LOTOS notation, this is given by:

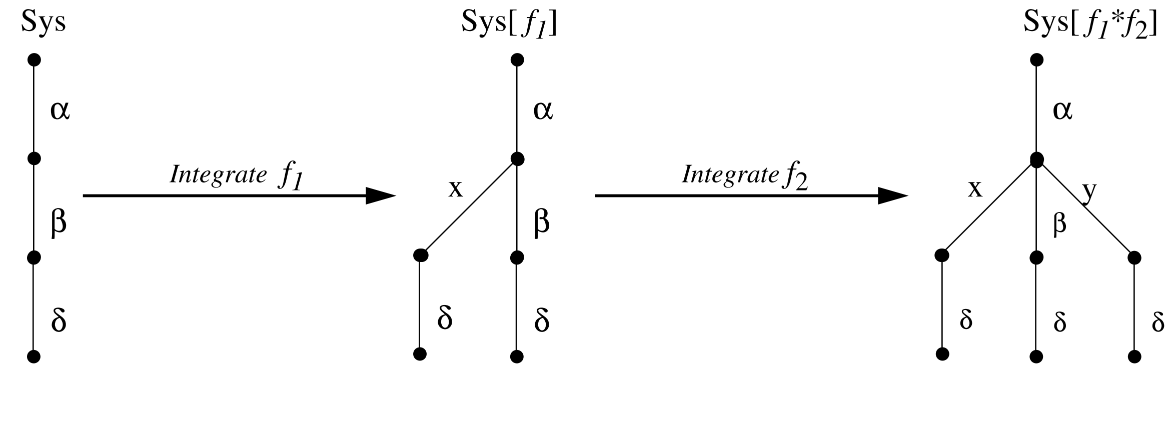


Fig. 5.7 Integration of f_1 and f_2 into Sys.

$$
\begin{aligned}
\mathrm{Tc} &= \mathrm{T}\ (\mathrm{Sys}[\mathrm{f1}\ |[]|\ \mathrm{f2}]) \\
&= \mathrm{T}\ (\alpha;\ (x;\ y;\ \textbf{exit}\ []\ \beta;\ \textbf{exit}\ []\ y;\ x;\ \textbf{exit})) \\
&= \alpha;\ (i;\ x;\ y;\ \textbf{exit}\ []\ \ i;\ \beta;\ \textbf{exit}\ []\ \ i;\ y;\ x;\ \textbf{exit})
\end{aligned}
$$

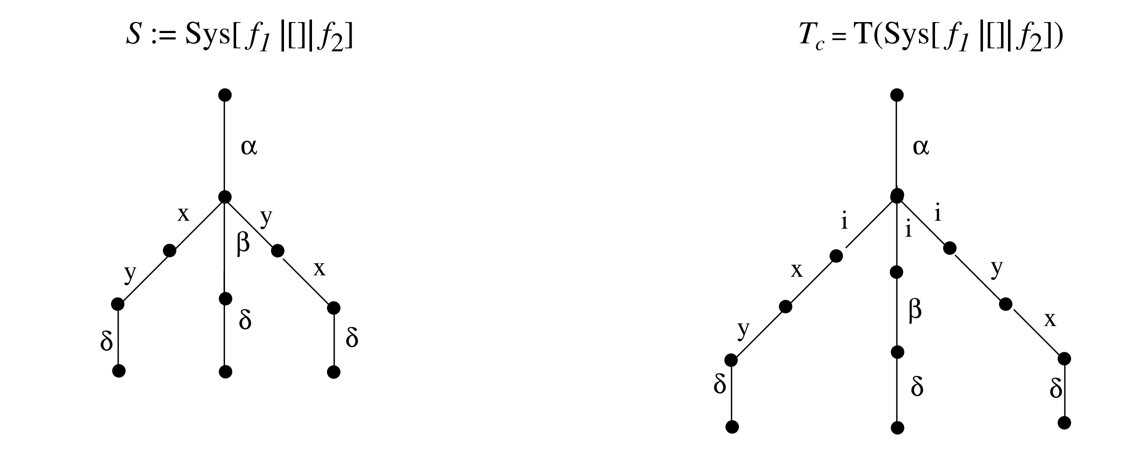


Fig. 5.8 A specification and its testing process

The next step is to express T_c as an irreducible set of test cases, from which a set of useful test cases are selected[BrTV91]. Two test cases T_1 and T_2 are shown in Figure 5.9. Executing these test cases against the integration of the two features result in deadlocks. This is an indication that an interaction between the two features exists, as shown in the figure.

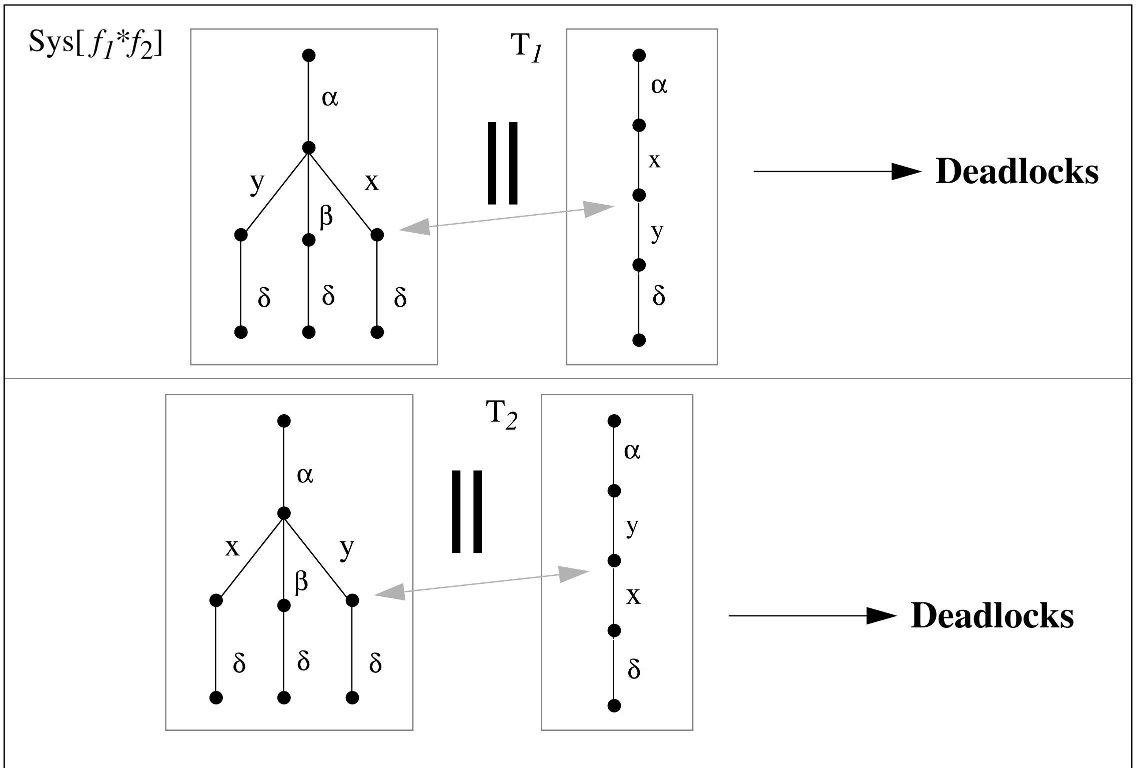


Fig. 5.9 Testing the Integration: Interactions are shown by deadlocks

5.2.4 Executing the System and Analysing the Results (step ❺ and ❻)

Execute the specification against the test suite to detect deadlocks. A deadlock implies one of the following:

(1) The two features are not intended to be activated simultaneously. Therefore, the deadlock is simply a confirmation of this fact,

(2) The two features are intended to be activated simultaneously. Therefore, the deadlock is interpreted as a feature interaction.

5.3 Application of the Methodology

In this section, we apply the methodology to a concrete telephone example. The purpose of this exercise is to check for interactions between two features: *Call Waiting (Cw)* and *Three Way Calling (Twc)*. For each feature, we give its informal description, followed by the structure of its formal specification, then we show how to integrate both features into POTS. Finally, we give a testing process which shows how the interaction is detected. For the sake of clear presentations, we will use LTSs instead of LOTOS processes to express the global behaviour of features. Only those traces which are pertinent to our example are shown. A detailed description of the specifications is given in [Hibo94].

5.3.1 Formal Specifications of Twc and Cw in the Context of POTS: Step ❶

The objective of this step is to produce the formal specifications of each of Three Way Calling and Call Waiting in the context of POTS. These results are shown in Figure 5.11 on page 80 and Figure 5.13 on page 82. Each specification is preceded by an informal description of the feature.

5.3.1.1 Informal Description of Twc

> *Twc is a feature that allows a station in the talking state to add a third party to the call without operator assistance. To add a third party to the call, the Twc customer flashes the switchhook once to place the other party on hold, receives recall dial tone, dials the third party's telephone number, and then flashes switchhook again to establish the TWC connection. The second switchhook flash may occur any time after the completion of dialing. After the TWC connection has been established, the customer with the service activated may disconnect the last party added by a single switchhook flash. The customer with the service activated may terminate the TWC by disconnecting. If either of the other two parties hangs up while the service-activating customer remains off-hook, the TWC is returned to a two-party connection between the remaining parties.*

5.3.1.2 Formal specification of Twc

Twc requires two legs[1] to be created within the same connection. A static structure is shown in Figure 5.10. The dynamic behaviour of the system is shown in Figure 5.11; part (a) models the behaviour of the first leg, where a user **A** establishes a connection with user **B**; part (b) shows the second leg of the connection, which can be created only after user **A** reaches a talking state.

In order to model the physical realities of existing telephone sets, the two legs are specified so that they synchronize on the action *flashhook*, which is used to transfer control from

1. A leg is a physical path between two users in a multi-party connection.

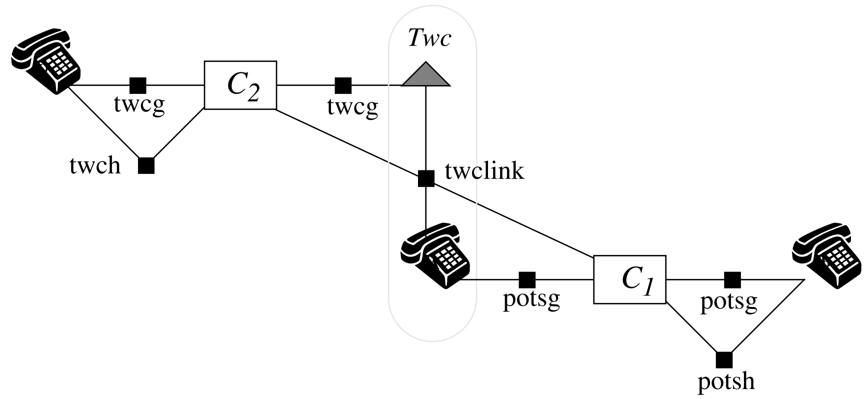


Fig. 5.10 Integrating Twc into POTS

one leg to another. When **A** reaches the talking state in (a), it may either continue talking to **B** or flash the hook. Before we continue with the description of the behaviour, please note that synchronization points between LTSs, such as C_1 and C_2, are shown by light lines, with double arrows. This notation is used in the rest of the thesis as well.

By flashing the hook, which is a synchronization point between the two legs, execution within the first leg is suspended while the second leg attempts to establish a second connection with **D**. Once the connection is established, a second flash hook resumes the execution of both legs, putting the system in a global state where **A** is talking to **B** within the first connection, and **A** is talking to **D** within the second connection. Our interpretation of such a global state is that a three way connection is formed between **A**, **B**, and **D**. A third flash hook terminates the last created connection (**A**, **D**). The system then returns to a global state where **A** is talking to **B**. From this state *Twc* can be initiated again to create a new three way connection involving **A**, **B**, and another user.

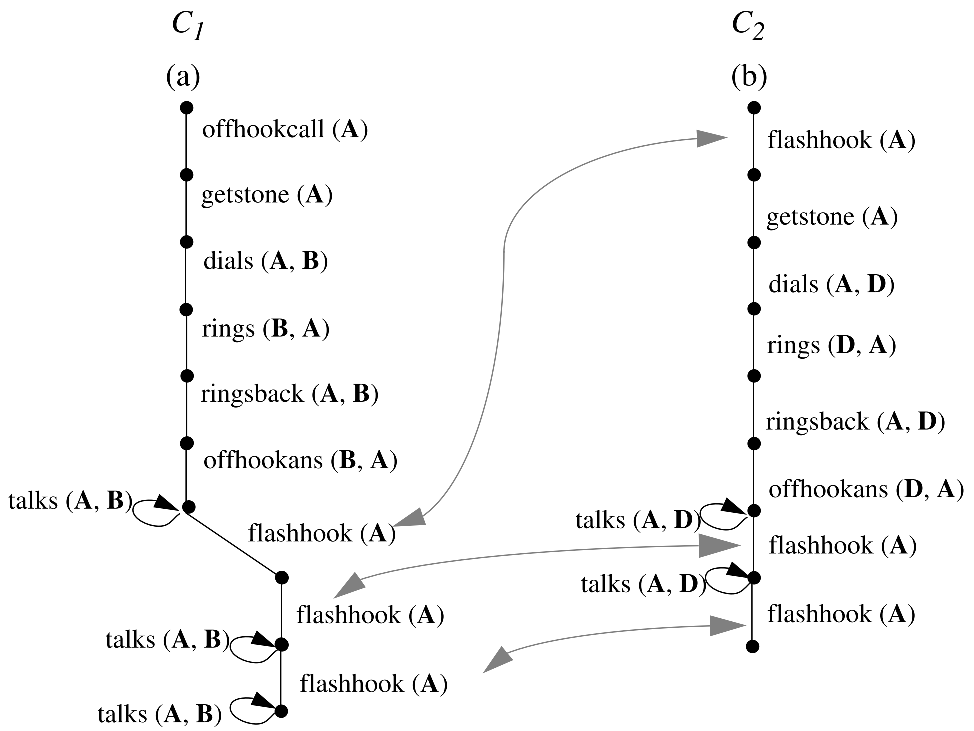


Fig. 5.11 LTSs of *Twc*. (a) first leg; (b) second leg.

5.3.1.3 Informal Description of Call Waiting (Cw)

Call Waiting (Cw) is a feature whereby a line in the talking state is alerted by a call waiting tone when another call is attempting to complete to that line. The call waiting tone is only audible to the line with the Cw feature activated. Audible ringing is returned to the originating line. The service also provides a hold feature that is activated by a flashhook. Consecutive flashes allow the customer (with the service activated) to alternately talk to the original and the newly calling party. If the customer with the service activated hangs up while one party is on hold, the customer with the service activated is automatically rung back, and upon answer is connected to the held party.

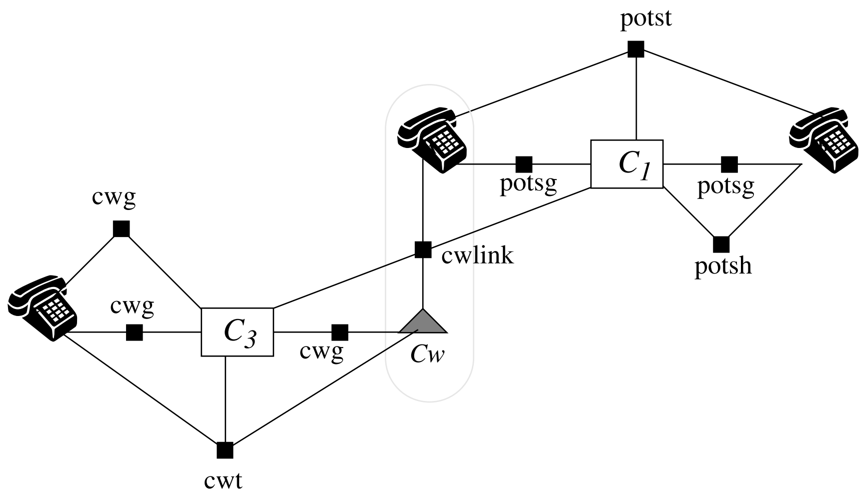


Fig. 5.12 Integrating Cw into POTS

5.3.1.4 Formal Specification of Cw

A typical trace of *Cw* would include a sequence of events which allows a user to establish a connection with a first user, and while in the talking state, respond to a second caller. Note that in real applications it is sometimes possible for a user to become connected to a waiting caller, by effect of a *flashhook*, even before a call waiting tone is received.

Assuming that user **A** is a subscriber of *Cw*, Figure 5.12 shows the structure of a connection between three users **A, B**, and **C**. The relevant aspects of the corresponding behaviour of C_1 and C_3 are given in Figure 5.13. It illustrates a scenario in which **A** may establish a talking session with user **B**, and then respond, using the *Cw* feature, to a third user **C**.

Assuming that the system has reached a global state where **A** is talking to **B** in C_1 and **C** has just finished dialling **A** in C_3, then synchronization between C_1 and C_3 may occur either on *cwtone* after the ***i***, in the left branch of C_3, or on *flashhook*, the second action after the ***i*** in the right branch. We model these two choices using the internal action ***i*** to express the fact that, sometimes, synchronization occurs first on *cwtone*, then on *flashhook*, and other times, synchronization occurs only on *flashhook*, without synchronizing on *cwtone*. The latter case occurs when C_3 sends a *cwtone* at the same time that C_1 flashes the hook. We interpret this in the following way. The *cwtone* is never received by C_1 and the *flashhook* sent by C_1 is interpreted by C_3 as a response to its *cwtone*. Since we abstract from the notions of send and receive, we model this (loss of *cwtone*) by an internal action ***i*** as shown in the right branch of

C_3. This is followed by a synchronization on the *flashhook* in the left branch of C_1 and the one in the right branch of C_3.

In the other case, when **A** receives the *cwtone* signal, it moves to a state where it may continue to talk to **B** or respond to user **C**. If **A** responds to **C**, by pressing the *flashhook* button, the connection between **A** and **B** is put on hold and a talking session between **C** and **A** is started. A second *flashhook* signal from **A** disconnects **A** from **C** and resumes the talking session between **A** and **B**. At this point, **A** may use the feature again to respond to another user. Note that the fact that **B** is put on hold is implied but is not explicitly shown in Figure 5.13. Also note

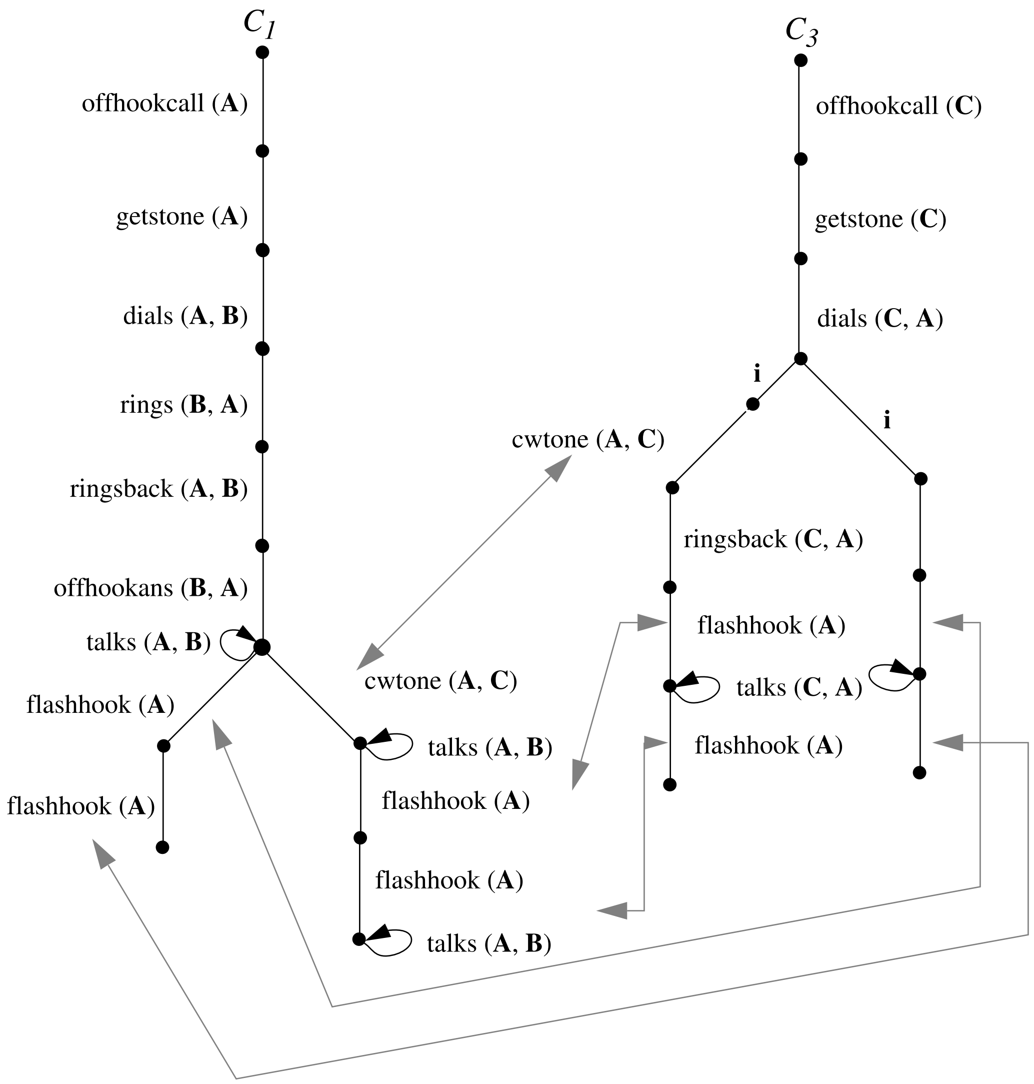


Fig. 5.13 LTS of *Cw* in the context of POTS

that we consider the activation of only one instance of *Cw* at a time. So, in the above illustration, once **A** activates *Cw* to talk to **C**, another user who attempts to call **A** would receive a busy signal. Finally, note that the four actions in the middle of the two branches of C_3 apply to both of them, for space restrictions.

5.3.2 Composition of Twc and Cw in the Context of POTS: Step ❷

Let the behaviours of the two features be expressed by the following two processes.

Twc [PotsGate,PotsTalk,PotsHang,TwcLink,TwcGate,TwcTalk,TwcHang]

and

Cw [PotsGate,PotsTalk,PotsHang, CwLink, CwGate,CwTalk,CwHang]

Then, their composition is given by the following behaviour expression:

```
behaviour
(
    Twc [PotsGate,PotsTalk,PotsHang,TwcLink,TwcGate,TwcTalk,TwcHang]
    |[PotsGate,PotsTalk,PotsHang]|
    Cw [PotsGate,PotsTalk,PotsHang,CwLink,CwGate,CwTalk,CwHang]
)
```

These two processes synchronize on all the events that occur on the gates associated with POTS and interleave on the rest of the gates. The labelled transition system representation of the whole system is quite large. Current tools for analysing LOTOS behaviours still lack the power and efficiency to generate such behaviour trees. Therefore, partial exploration of the tree using the step by step execution mode remains the best possible approach. The tree in Figure 5.14 was generated using this approach. The tree is read in the following manner. The root of the tree is on the top left hand side of the page. From the root there is a choice between the first action (line 1), which at the top of the page, and another action (not shown) beyond the bottom of the page. Similarly, there is a choice between the two actions at lines 11 and 22. The tree shows three types which are sufficient to give a general idea about the resulting behaviour of the composition:

(1) all the traces which can be executed by the *Twc* process alone, in the context of POTS;

(2) all the traces which can be executed by the *Cw* process alone, in the context of POTS;

| PotsGate1 !**A** !offhooktocall;
| | PotsGate1 !**A** !getstone;
| | | PotsGate1 !**A** !dials !**B**;
| | | | PotsGate1 !**B** !rings !**A**;
| | | | | PotsGate1 !**A** !ringsback !**B**;
| | | | | | PotsGate1 !**B** !offhookans !**A**;
| | | | | | | PotsTalk1 !**A** !talks !**B**;
| | | | | | | | TwcLink !**A** !flashhook;
| | | | | | | | | TwcGate !**A** !getstone;
| | | | | | | | | | TwcGate !**A** !dials !**D**;
| | | | | | | | | | | TwcGate !**D** !rings !**A**;
| | | | | | | | | | | | TwcGate !**A** !ringsback !**D**;
| | | | | | | | | | | | | TwcGate !**D** !offhookans !**A**;
| | | | | | | | | | | | | | TwcTalk !**A** !talks !**D**;
| | | | | | | | | | | | | | | TwcLink !**A** !flashhook;
| | | | | | | | | | | | | | | | PotsTalk1 !**A** !talks !**B**;
| | | | | | | | | | | | | | | | | TwcTalk !**A** !talks !**D**;
| | | | | | | | | | | | | | | | | | TwcHang !**D** !hangsup !**A**;
| | | | | | | | | | | | | | | | | | | PotsHang1 !**A** !hangsup;
| | | | | | | | | | | | | | | | | | | | PotsHang1 !**B** !hangsup;
| | | | | | | | | | | | | | | | | | | | | exit
| | | | | | | | | | | CwGate !**C** !offhooktocall;
| | | | | | | | | | | | CwGate !**C** !getstone;
| | | | | | | | | | | | | CwGate !**C** !dials !**A**;
| | | | | | | | | | | | | | CwLink !**A** !cwtone !**C**;
| | | | | | | | | | | | | | | CwGate !**C** !ringsback !**A**;
| | | | | | | | | | | | | | | | CwLink !**A** !flashhook;
| | | | | | | | | | | | | | | | | TwcGate !**D** !rings !**A**;
| | | | | | | | | | | | | | | | | | CwTalk !**A** !talks !**C**;
| | | | | | | | | | | | | | | | | | | CwHang !**C** !hangsup;
| | | | | | | | | | | | | | | | | | | | CwLink !**A** !flashhook;
| | | | | | | | | | | | | | | | | | | | | TwcGate !**A** !ringsback !**D**;
| | | | | | | | | | | | | | | | | | | | | | TwcGate !**D** !offhookans !**A**;
| | | | | | | | | | | | | | | | | | | | | | | TwcTalk !**A** !talks !**D**;
| | | | | | | | | | | | | | | | | | | | | | | | TwcLink !**A** !flashhook;
| | | | | | | | | | | | | | | | | | | | | | | | | PotsTalk1 !**A** !talks !**B**;
| | | | | | | | | | | | | | | | | | | | | | | | | | TwcTalk !**A** !talks !**D**;
| | | | | | | | | | | | | | | | | | | | | | | | | | | TwcHang !**D** !hangsup !**A**;
| | | | | | | | | | | | | | | | | | | | | | | | | | | | PotsHang1 !**B** !hangsup;
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | PotsHang1 !**A** !hangsup;
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | exit
| | | | | | | | CwGate !**C** !offhooktocall;
| | | | | | | | | CwGate !**C** !getstone;
| | | | | | | | | | CwGate !**C** !dials !**A**;
| | | | | | | | | | | CwLink !**A** !cwtone !**C**;
| | | | | | | | | | | | CwGate !**C** !ringsback !**A**;
| | | | | | | | | | | | | CwLink !**A** !flashhook;
| | | | | | | | | | | | | | CwTalk !**A** !talks !**C**;
| | | | | | | | | | | | | | | CwHang !**C** !hangsup;
| | | | | | | | | | | | | | | | CwLink !**A** !flashhook;
| | | | | | | | | | | | | | | | | PotsTalk1 !**A** !talks !**B**;
| | | | | | | | | | | | | | | | | | PotsHang1 !**A** !hangsup;
| | | | | | | | | | | | | | | | | | | PotsHang1 !**B** !hangsup;
| | | | | | | | | | | | | | | | | | | | exit
| | | | | | | | | | | TwcLink !**A** !flashhook;
| | | | | | | | | | | | TwcGate !**A** !getstone;
| | | | | | | | | | | | | TwcGate !**A** !dials !**D**;
| | | | | | | | | | | | | | CwLink !**A** !cwtone !**C**;
| | | | | | | | | | | | | | | CwGate !**C** !ringsback !**A**;
| | | | | | | | | | | | | | | | CwLink !**A** !flashhook;

Fig. 5.14 Some traces from the composition of *Cw* and *Twc*

(3) all the traces which synchronize on the common POTS actions and interleave on the independent actions of *Cw* and *Twc*.

The first trace, lines (1 to 21), is of type (1). The second trace, lines (1 to 7; 42 to 54) is of type (2), where the ";" is read followed by. Finally, the two traces which consist of lines (1 to 10; 22 to 41) and lines (1 to 7; 42 to 44; 55 to 60) are of type (3).

5.3.3 Integration of Twc and Cw into POTS: Step ❸

From a structural point of view, Figure 5.15 represents the integration of the two features into the POTS specification. The figure is composed of three core processes: process C_1 represents a modified version of the original POTS controller so that it supports basic connections as well as the *Twc* and *Cw* features; process C_2 handles the second leg of a three way call connection; and finally, process C_3 handles a call waiting connection.

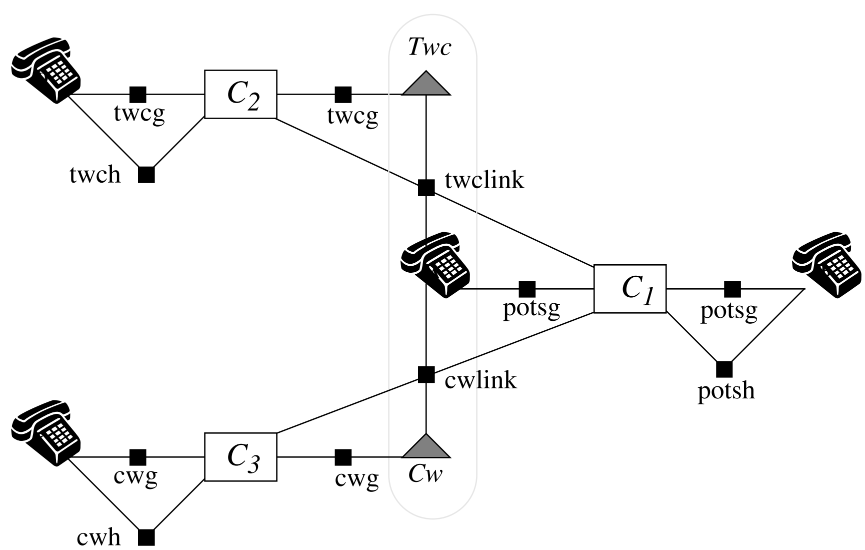


Fig. 5.15 Integrating Twc and Cw into POTS

The integration of the features into POTS is always associated with assumptions. For the proper application of our methodology, we distinguish two kinds of assumptions. The first kind, which is an integral part of the methodology itself, requires that the integration of the features be carried out without any knowledge of what effects the behaviour of each feature may have on the other. The second kind of assumptions is specific to the behaviour of each feature and how it is integrated into POTS; some of these assumptions have a direct influence on whether or not an interaction with other features manifests itself. To demonstrate the generality of the approach, we will consider two cases where different assumptions are made about the integration of the two feature into POTS.

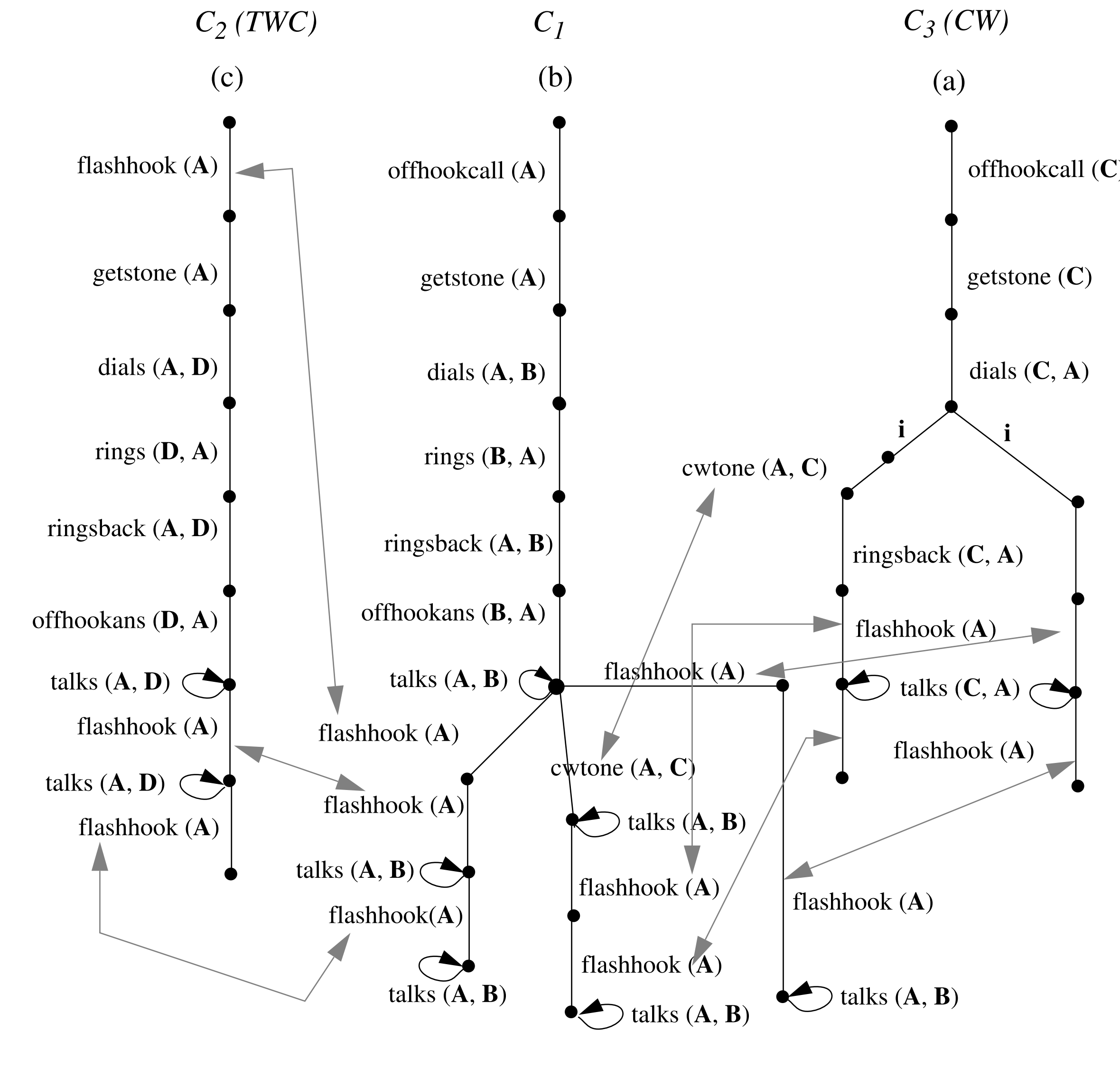


Fig. 5.16 LTSs of Twc and Cw in the context of POTS: Case 1

Case 1: We consider that user **A** has a single flash hook button, which is used to invoke both *Cw* and *Twc*. This case shows how an interaction occurs due to the interpretation of the flash hook signal in the wrong context. A partial behaviour of the integration, under this assumption, is shown in Figure 5.16. Note that the behaviour of C_2 in this figure is the same as that of C_2 in Figure 5.11 on page 80 and the behaviour of C_3 is the same as that of C_3 in

Figure 5.13 on page 82. However, the behaviour of C_1 combines the behaviours of C_1 in Figure 5.11 and C_1 in Figure 5.13. So, after *offhookans (**B**, **A**),* in this figure, C_1 may synchronize with both C_2 and C_3, when a common synchronization point such as *flashhook* is reached. Synchronization events between C_1 and each of C_2 and C_3 are shown by dashed lines in the LTS trees.

Assuming that user **A** has reached the talking state with **B** in Figure 5.16 (b), then the following actions are possible, depending on the state of C_3:

- If C_3 moves to the state after the ***i*** in the left branch, then the system moves to a global state where C_1 may synchronize with C_3 on the action *cwtone*. If this occurs, the system moves to a new state where **A** is still talking to **B** in behaviour C_1 and C_3 is waiting to synchronize on a *flashhook* signal to continue with its sequence of actions. Therefore, in this context, a flash hook signal has only one interpretation, which is that of *Cw*. Note, however, that *Twc* cannot be activated after this point, which introduces an interaction similar to that of case 2.
- If C_3 moves to the state after the ***i*** in the right branch, then the next synchronization point between C_1 and C_3 is the *flashhook* signal. However, from this state, C_1 may also synchronize with C_2 on the same signal. Therefore, two different interpretations of the *flashhook* are possible, one in the context of *Twc* and the other in the context of *Cw*.

Case 2: Even if we assume that user **A** can somehow distinguish[1] between the contexts of the flash hook signals, the two features may still interact. The behaviour of the integration, under this assumption, is shown in Figure 5.17. Note that the behaviour of C_2 is the same as its counterpart in Figure 5.16, whereas the behaviours of C_1 and C_3 are simplified, because in this case the ambiguity concerning the *flashhook* is removed from the integration. So, after *offhookans (**B**, **A**)*, C_1 may synchronize either on *cwtone* or *flashhook* which, in this case, has a *Twc* interpretation only.

Since both features require a second leg to establish their respective connection involving three users, and since there is only one such leg then one feature is automatically disabled once the second leg is seized by the other feature. As just stated, **A** has total control of

1. We may think of **A** as having two different flash hook buttons to invoke *Cw* and *Twc*.

which feature he/she wants to use. To use *Cw*, the user waits for *cwtone* signal, then presses the flash hook button associated with *Cw*. So, in the context of *Cw*, the state from which the *flashhook* signal becomes possible can be reached only by executing the sequence starting with *cwtone*, which is the right branch in Figure 5.17 (b). To use *Twc*, the user executes the sequence

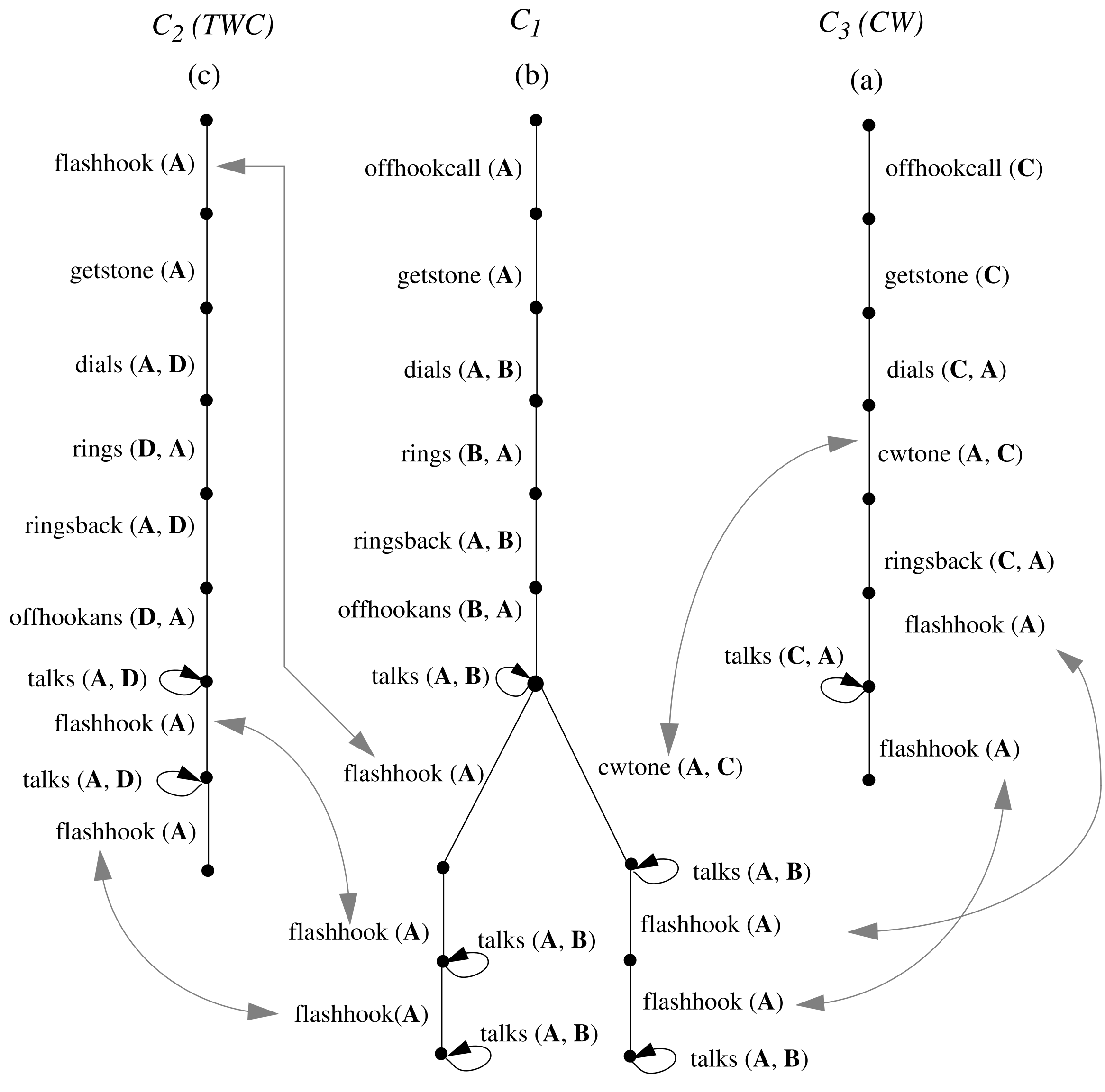


Fig. 5.17 LTSs of Twc and Cw in the context of POTS: Case 2.

starting *flashhook*, in the left branch.

Notice that the choice between the actions of *Twc* and *Cw* is deterministic and no confusion arises with respect to the flash hook. However, an interaction still exists because, once one of the two features is activated, the other one is automatically disabled, due to the limited resource of legs.

5.3.4 Generation of Test Cases: Step ❹

In order to completely test the system, one would need to generate the canonical tester from the composed specification of step ❷ above. The canonical tester is then used to derive the set of all test cases, which are executed against the integration of the two features. However, it is a well known problem that systems of this size cannot be tested exhaustively. Therefore, we used our knowledge and experience to derive some useful test cases directly from the composition without going through the canonical tester. Of course, all the test cases we have derived would have been generated from the canonical tester, because all the traces of a specification are traces of its canonical tester.

5.3.5 Detection of Interactions between Twc and Cw: Step ❺

In order to understand the kind of interactions that occurs between these two features and how to detect them, let us use a typical communication scenario for each of the two cases that we described in Section 5.3.3 on page 85.

Case 1: Suppose that C_1 has reached the talking state according to the LTS of Figure 5.16 (b). Next, suppose that C_3 has reached the state just after the *dials(**C**, **A**)* action. From this state, C_3 may non-deterministically choose the left branch or the right branch. For this case, we are interested in the right branch. Then C_3 may execute the *ringsback(**C**, **A**)*, which requires no synchronization with C_1, and becomes ready to execute the next action, which is *flashhook(**A**)*. At this point, process C_1 is in a state where it is able to synchronize either on *flashhook(***A***)* with C_2, because it is in a state where this action may be executed or on *flashhook(***A**) with C_3, because this is the next action according to the specification. This ambiguity, which is related to the interpretation of the flash hook signal in two different contexts, creates a situation which may produce unexpected results for user **A**.

Formally, we detect this ambiguity by composing the behaviour of the integrated features in parallel with the testing process such as the one shown in Figure 5.18. The intention of the *flashhook* on line 55, if we abstract from LOTOS gates, is to invoke the *Twc* feature. However, if process C_3 executes the sequence of events:

```
process Testing_Cw_Twc
: exit:=
1   offhookcall(A);
2   getsTone(A);
3   dials(A, B);
4   rings(B, A);
5   ringsback(A, B);
6   offhookans(B, A);
7   talks(A, B);
42  offhookcall(C) ;
43  getstone(C);
44  dials(C, A);
55  flashhook(A);
56  getstone(A);          Deadlock (Case 1)
57  dials(A, D);
58  cwtone(A, C);
59  ringsback(C, A);
60  flashhook(A);
61  stop
endproc
```

Fig. 5.18 Testing process for *Twc* and *Cw*: Case 1

*offhooktocall(**C**); getstone(**C**); dials(**C**,**A**); i; ringsback (**C**, **A**)*; *flashhook(**A**)*

then the next action *getstone(**A**)*, from the test process does not have a match in C_3, which results in a deadlock on this action of the tester. It is important to recall that, if non-determinism is present, execution of a testing process may not always lead to the same results. For this reason, we must execute the same test several times to detect a possible deadlock, and there is no guarantee that the deadlock will ever be detected by such repeated executions.

Case 2: This is similar to the previous case, except that when user **A** reaches a talking state with user **B**, as shown in Figure 5.17 (b), a choice must be made between the invocation of the *Twc* and *Cw*. Although the choice is deterministic, only one feature is allowed to execute for any given connection. Executing this specification with the same testing process results in a deadlock as well, as shown in Figure 5.19, but for a different reason, as explained in the next section.

Lines 1 to 7 of the tester synchronize with C_1 to establish a connection between **A** and **B**. Lines 42 to 44 synchronize with C_3. Line 55 offers *flashhook* signal to establish a three way call. The next two actions of the tester, lines 56 and 57, would synchronize with C_2. Finally, the tester deadlocks on the *cwtone* action, which is not offered by this integration from its current state. If the two features were able to execute simultaneously, the integration would offer this action from the current state, and no deadlock occurs at this point.

```
      process Testing_Cw_Twc
      : exit:=
1     offhookcall(A);
2     getstone(A);
3     dials(A, B);
4     rings(B, A);
5     ringsback(A, B);
6     offhookans(B, A);
7     talks(A, B);
42    offhookcall(C);
43    getstone(C);
44    dials(C, A);
55    flashhook(A);
56    getstone(A);
57    dials(A, D);
58    cwtone(A, C);  <-------------- Deadlock (Case 2)
59    ringsback(C, A);
60    flashhook(A);
61    stop
      endproc
```

Fig. 5.19 Testing process for *Twc* and *Cw*: Case 2

5.3.6 Analysing the Results: Step ❻

Under the first assumption, where we model the lack of synchronization on the call waiting signal, the problem is explained in terms of an ambiguous global state where both features have different interpretation of the flash hook signal with respect to the user on which the features are active. Flashing the hook at the wrong time forces user **A** to establish a call waiting connection, even though the user has flashed the hook with the intention of establishing a three way call connection.

Under the assumption of the second case, where the loss of signals is not taken into consideration, our analysis of the deadlock reveals that the integration of the two features, in the context of POTS, assumes that each of the features has exclusive use of the second leg which is used to complete both a three way communication session and a call waiting communication session. Therefore, an interaction between these two features is due to this limited resource.

CHAPTER 6 Specifying Features and Detecting their Interactions: Case Studies

In this Chapter, we apply our methodology to a number of pair-wise feature interactions [CGLN94] [Lata91]. In our case studies, we deal with two types of interactions only: Single User Single Element (SUSE) and Multiple User Single Element (MUSE). In SUSE, we detect the interactions that occur between the features of the same user; in MUSE, we detect the interactions between the features of two different users. For each example, we give an informal description of the features, followed by their formal specifications, then we show how to integrate them into POTS, and finally, we give a testing process which shows how the interaction is detected.

6.1 Notations and Remarks

In the rest of this Chapter, we adapt the following *conventions*, when talking about a feature. The generic reference to a feature is abbreviated by its name; the LOTOS process representing its behaviour is parameterized by the user, written in ***bold italic***, on which the feature is active, along with any other required parameters, written in *italic*. For example, if **A** subscribes to a *Call Forward Always* feature where user **C** is the new destination, then we say that **A** subscribes to *Cfa* and the process *Cfa(**A**, C)* acts on behalf of user **A**. Also, for processes which manipulate sets such as *Ocs*, the notation *Ocs(**A**)* will be used to refer both to the process and to the set, when no confusion arises from the context. For the features discussed in this Chapter, we use the notations of Figure 6.1.

6.1.1 Remark 1: Structure of Specifications

For each specification in this Chapter, a graphical representation of the specification's structure, showing the top level processes, is given. No attempt is made to show the details of

Feature	Process Notation	Meaning
Call Waiting	*Cw(**A**)*	**A** subscribes to *Call Waiting*
Call Forwarding Always	*Cfa(**A**, B)*	Calls directed to **A** are always forwarded to *B*
Call Forward on Busy Line	*Cfbl (**A**, B)*	When **A** is busy, her/her incoming calls are forwarded to *B*
Three Way Calling	*Twc(**A**)*	**A** subscribes to *Three Way Calling*
Terminating Call Screening	*Tcs(**A**)*	The set of users who are in the terminating screening list of **A**
Originating Call Screening	*Ocs(**A**)*	The set of users who are in the originating screening list of **A**
Distinctive Ringing	*Dsr(**A**, B)*	*B* shares the same physical line as the main user **A**, but each user has a distinctive ringing patterns
Unlisted Number	*Uln(**A**)*	**A** subscribes to *Unlisted Number*
Calling Number Delivery	*Cnd(**A**)*	**A** subscribes to *Calling Number Delivery*
Automatic Call Back	*Acb(**A**, B)*	When **A** calls *B* who is busy, **A** is called back when *B* becomes idle
Automatic ReCall	*Arc(B, **A**)*	If **A** is busy when called by *B, B* is recalled when **A** becomes idle

Fig. 6.1 Notations for Features in Chapter 6

how processes communicate with each other or which gates they use; only those gates which are necessary for the understanding of a specification are shown. Also, note that the term *structure* in this Chapter always refers to the structure of a specification.

6.1.2 Remark 2: Implicit Reference to the Global Constraint Process

The global behaviour of all specifications in this Chapter assume the implicit existence of the global constraints process, as described in Chapter 4. Where not strictly required for the understanding of the example at hand, the global constraint process may be mentioned but no effort is made to explicitly describe its behaviour.

6.1.3 Remark 3: Generation of Testing Processes

We have argued in Section 5.3.4 on page 89 that the generation of complete test suites and the selection of the most useful test sequences remains a challenging area of research. For the examples in this Chapter, we focus more on how to detect an interaction between two given features than on how to generate and select test cases.

For each example, only one or two test cases are derived for illustration purposes. Since we have not constructed the compositions of the features in this Chapter, our derivation of tests is based on the composition $POTS[f_1|[]|\,f_2]$ of partial behaviours of f_1 and f_2. The detection of interactions is explained in terms of the deadlocks that occur between the integration $POTS[f_1*f_2]$ and the testing processes.

To abstract from LOTOS code, we will continue to use LTSs. This implies that synchronizations between trees will be shown in terms of double headed arrows in the figures, supported by textual explanations, where appropriate. Also, for all the examples, we will consider a test case only up to the point where the deadlock is detected. In other words, if a deadlock would occur on action ***b*** in the test case ***a; b; c; exit***, we will present our test case as being ***a; b; stop***, and the results are explained in terms of the expression *(I* || ***(a;b;stop)****)*.

6.2 Case Studies

6.2.1 Example 1: Call Waiting and Call Forward on Busy Line

6.2.1.1 Description of Call Waiting (Cw)

Cw was introduced in Section 5.3.1.4 on page 81. For convenience, Figure 6.2 shows a simplified version of the behaviour shown in Figure 5.13 on page 82. The corresponding structure of the specification is shown in Figure 6.3.

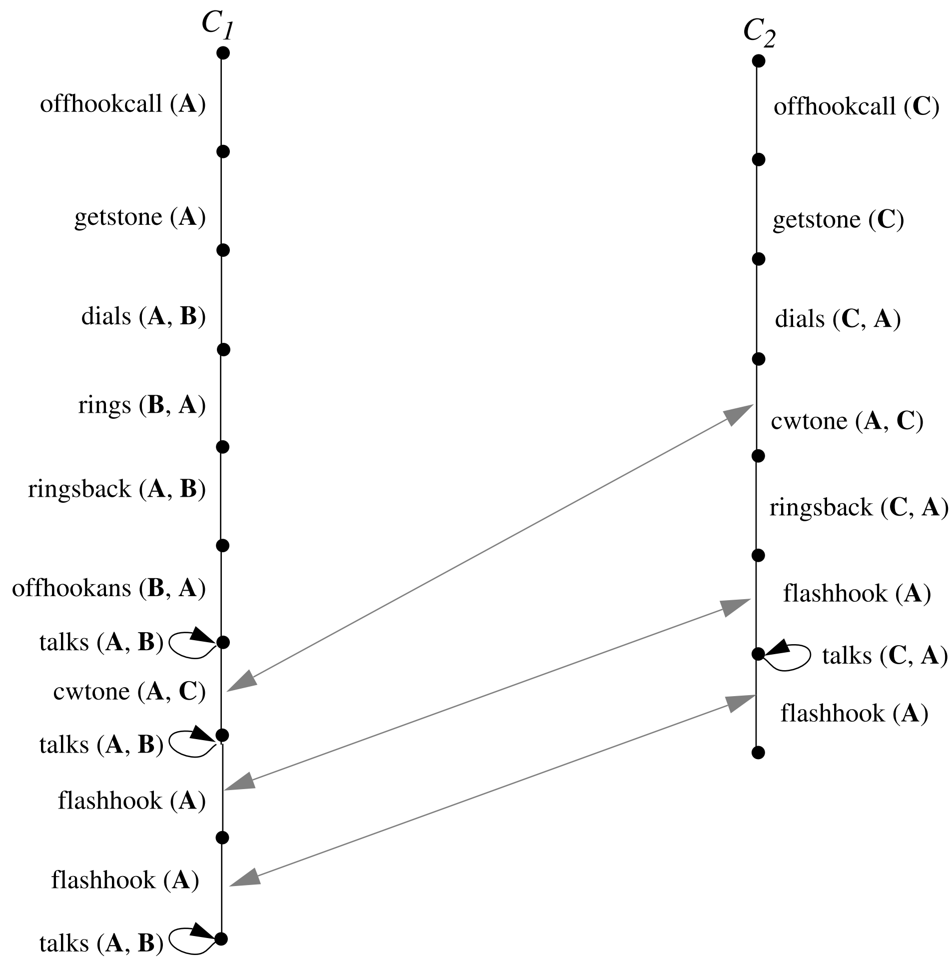


Fig. 6.2 LTS of *Cw* in the context of POTS

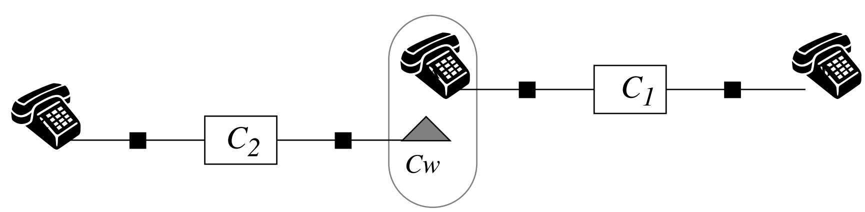


Fig. 6.3 Integrating Cw into POTS

6.2.1.2 Informal Description of Cfbl

Call forward busy line (Cfbl) provides the capability to associate with a given line another line to which calls should be forwarded when the given line is busy. The currently active connection is not affected in any way.

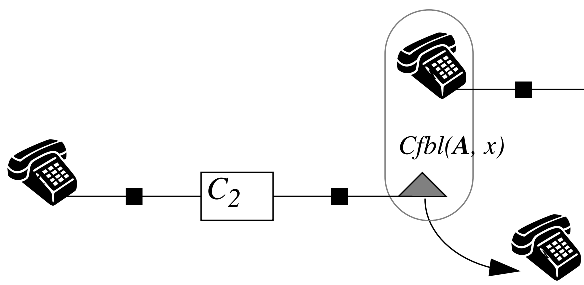


Fig. 6.4 Integrating Cfbl into POTS

6.2.1.3 Formal Specification of Cfbl

The *Cfbl* feature may be activated either by a calling side or a called side in a POTS connection. For a caller side, the feature becomes active as soon as the *offhookcall* event occurs. For a called side, the feature becomes active as soon as the *ringsfrom* event occurs. For illustration purposes, let us assume that **A** is a POTS caller who has activated *Cfbl(***A***, x)* as shown in Figure 6.4. The corresponding behaviour, consisting of C_1 and C_2, is shown in Figure 6.5. As soon as the *offhookcall* event occurs on **A** in C_1, a call from another user such as **C** to **A** will terminate at *x*, as shown in C_2. When user **C** initiates a connection, he/she starts as a basic POTS connection. However, once the dialled number is identified as **A**, **C**'s controller synchronizes with *x*, the process to which the call is forwarded instead of synchronizing with **A**, the user for which the call is originally intended. This is expressed by the *ringsfrom(x*, **C)** action which occurs on user *x* rather than **A**, with the assumption that C_2 interleaves with C_1 in execution, but the *offhookcall*(**A**) action in C_1 is executed before the *ringsfrom(x*, **C**) in C_2. If **A** is idle when the *dials(C, A)* action is executed, **A** rings and the call is not forwarded.

An analogous description would apply if user **B**, in Figure 6.4, has activated *Cfbl(***B***, x)* instead of **A**. In this case, C_1 in Figure 6.5 would remain the same, **B** would replace **A** in C_2, and the following scenario becomes valid. An incoming call from **C** would be forwarded to x only if **B** has gone past the action *ringsfrom(***B**, **A**) in the execution sequence of C_1. The assumption that C_1 and C_2 would interleave in execution remains.

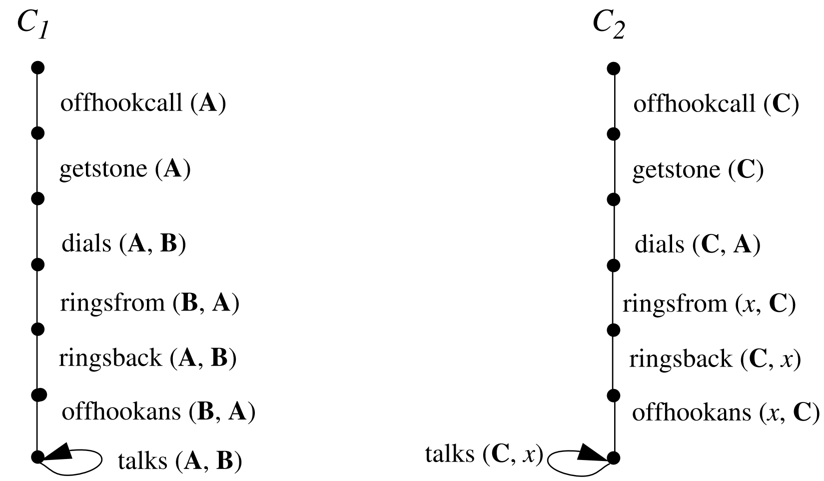


Fig. 6.5 LTS of *Cfbl* in the context of POTS

6.2.1.4 Integration of Cw and Cfbl into POTS

Figure 6.6 shows the structure of the specification after the integration of *Cw* and *Cfbl* into POTS. In this figure, we distinguish two core processes, C_1 and C_2. Process C_1 is obtained by modifying the POTS controller in order to support the functionality of *Cw*. Process C_2 is obtained by two separate modifications of the POTS controller, followed by a merge of the two behaviours, in order to support *Cw* and *Cfbl*. Figure 6.7 shows the behaviour of *POTS[Cw*Cfbl]*, which is expressed by C_1 and C_2. These two processes synchronize on events which are specific to *Cw* such as *cwtone* and *flashhook*, as shown by the dotted lines. We assume that **A** subscribes to both *Cw* and *Cfbl*. We also implicitly assume that users **A** and **C** play the role of a caller and user **B** plays the role of a called.

From a specification point of view, *Cw* can be integrated into POTS at a point where user **A** has reached a talking state within his/her first connection. One such point is after the action *offhookans (***B**, **A***)* in C_1, which moves the connection to a talking state. On the other hand, *Cfbl(A, x)* is allowed to become active at any point after the execution of the action

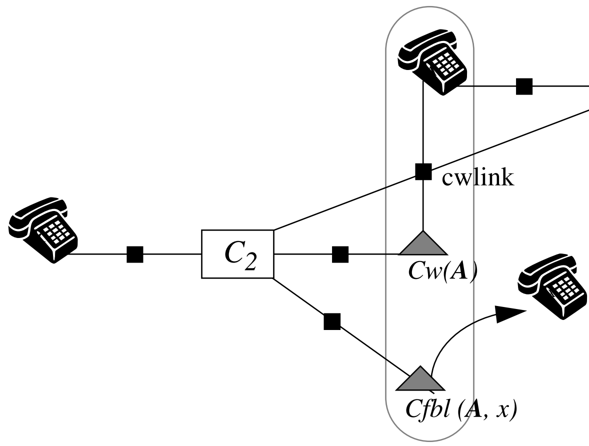


Fig. 6.6 Integrating Cfbl and Cw into POTS

*offhookcall(***A**). It is interesting to note the role of the global constraints in this case, through which the *Cfbl(***A***, x)* feature learns that user **A** is in a busy state. Detecting the interaction between these two features is described in the next section.

6.2.1.5 Detection of Interactions between Cw and Cfbl

The following is a typical communication scenario. User **A** establishes a connection with **B**, and while they are talking, user **C** attempts to call **A**. Since both features are designed to execute their first actions based on the busy state of the user on which they are active, and since the integration of the first feature does not take into account the side effects of integrating the second feature, then in the above scenario, one of the two features is non-deterministically activated, since they are both ready to execute their first actions, which are *cwtone(***A**, **C***)* and *ringsfrom(x*, ***C****)* in C_2. The internal action ***i*** is used to model the non-deterministic choice between these two actions. Note that the right branch of C_2 has no correspondent in C_1 because in that case C_1 simply remains in the talking state until a hang up occurs (not shown).

Figure 6.8 on page 100 shows a partial behaviour of the composition *POTS[Cw|[]|Cfbl]*, which is used to generate two testing processes, t_1 and t_2. In this composition, *Cw* and *Cfbl* synchronize on the actions *offhookcall(****C****)*, *getstone(****C****),* and *dials(****C, A****)*, which they share with POTS, and interleave on their independent actions *cwtone(****A, C****)* and *ringsfrom (x****, C****)*. By executing the behaviour tree of Figure 6.7 against either of t_1 or t_2, we reach a deadlock regardless of which branch C_2 executes, provided that **A** is in a talking state. For example, with respect to t_1, the system $I||t_1$ deadlocks on the action

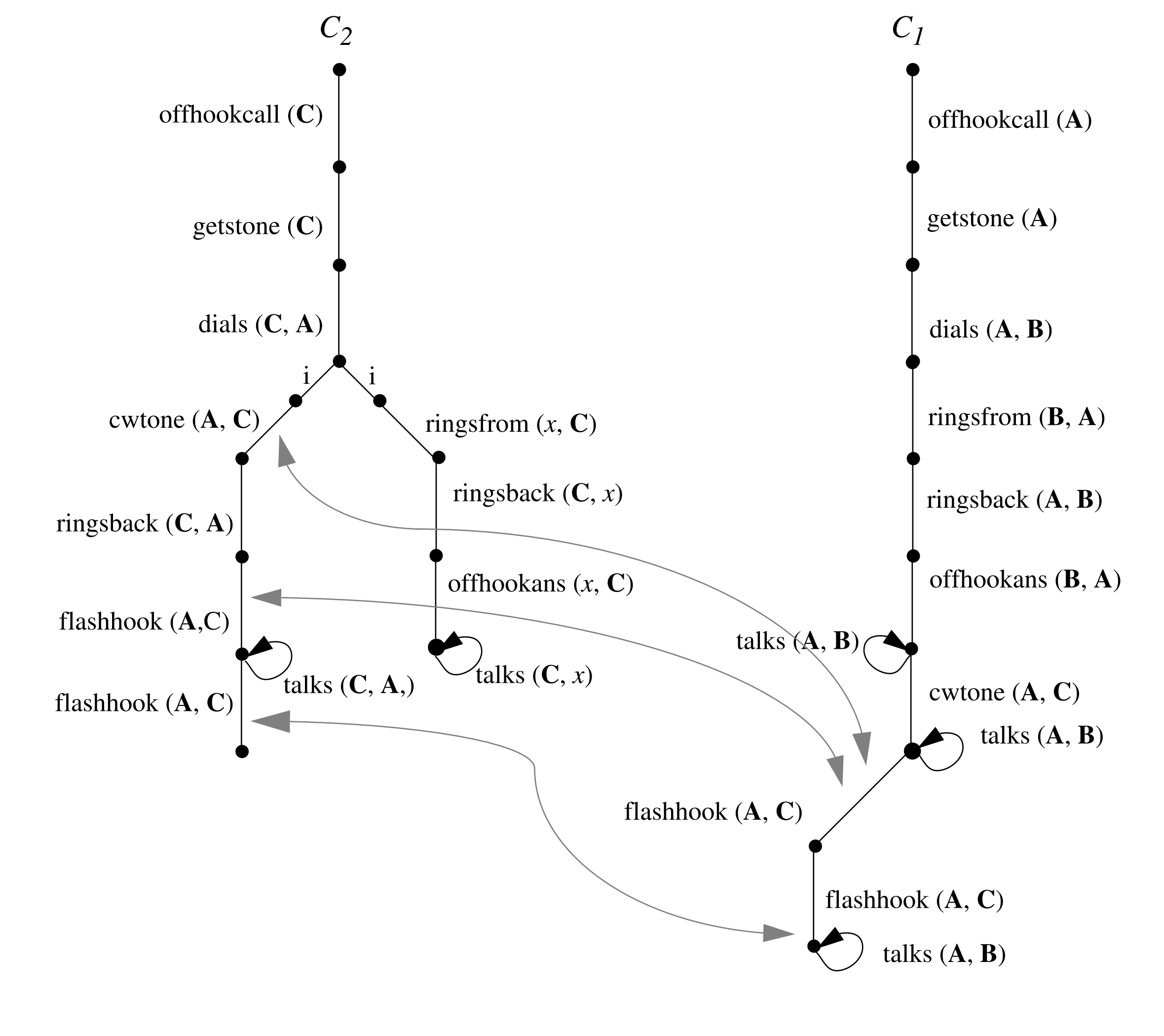


Fig. 6.7 LTSs of the integration *POTS[Cw*Cfbl]*

*ringsfrom(**x, C**)* if C_2 executes the right branch and deadlocks on *ringsback(**C, A**)* if it executes the left branch. This indicates that the two features cannot operate simultaneously. A similar result is obtained by using t_2.

A possible solution for this interaction is to establish a priority between these two features and inform subscribers of this decision. Another alternative is to leave this decision up to each subscriber.

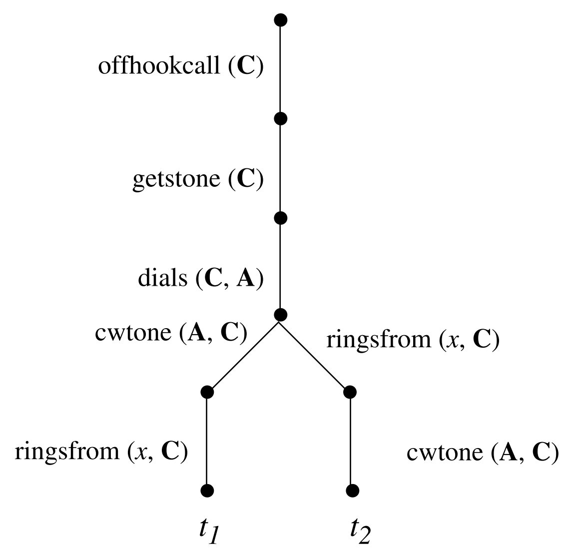


Fig. 6.8 Partial behaviour of the composition *POTS[Cw|[]|Cfbl]*

6.2.2 Example 2: *Automatic Recall and Terminating Call Screening*

6.2.2.1 Informal Description of Automatic Recall

*Automatic ReCall (Arc) is a feature which allows a subscriber **A** to setup his telephone set so that, if an incoming call arrives from another user **D** while **A** is busy, the call from **D** is memorized and an attempt is made to return the call when **A** becomes idle. If **D** is busy when the Arc feature attempts to return the call, the call is queued until both parties are idle. When this occurs, **A** is given a ringback. When the ringback is answered, a ring is sent to **D**.*

6.2.2.2 Formal Specification of Arc

Figure 6.9 shows the static structure of integrating *Arc*(***A***) into a POTS specification. The corresponding global behaviour of the system, shown in Figure 6.10, is composed of the behaviours C_1, C_2, and the global constraints process. In this figure, we assume that **A** subscribes to *Arc*, and while busy with **B**, it receives a call from **D**. Also, note that C_1 and C_2 do not communicate directly with each other. Communication between these two components is achieved by way of the global constraints.

The behaviour tree which corresponds to C_2 is obtained by modifying the behaviour of a

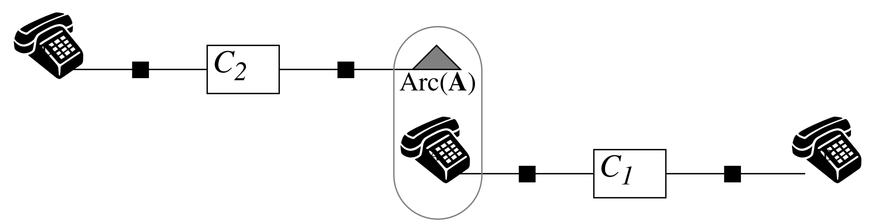


Fig. 6.9 Integrating *Arc* into POTS

POTS controller. From its initial state, C_2 offers to synchronize on two actions: *offhookcall(**D**)* which allows **D** to make a new call and *arcringsfrom*(**A**, ***D***) which allows **A** to be informed that an incoming call was attempted while **A** was busy. Similarly, when C_1 is in its initial state, it offers to synchronize on the action *offhookcall*(**A**). Therefore, from a global point of view, the two initial local states form a global state which allows all the actions that are possible from its independent constituents. For example, if C_1 executes the action *offhookcall(**A**)* and C_2 follows

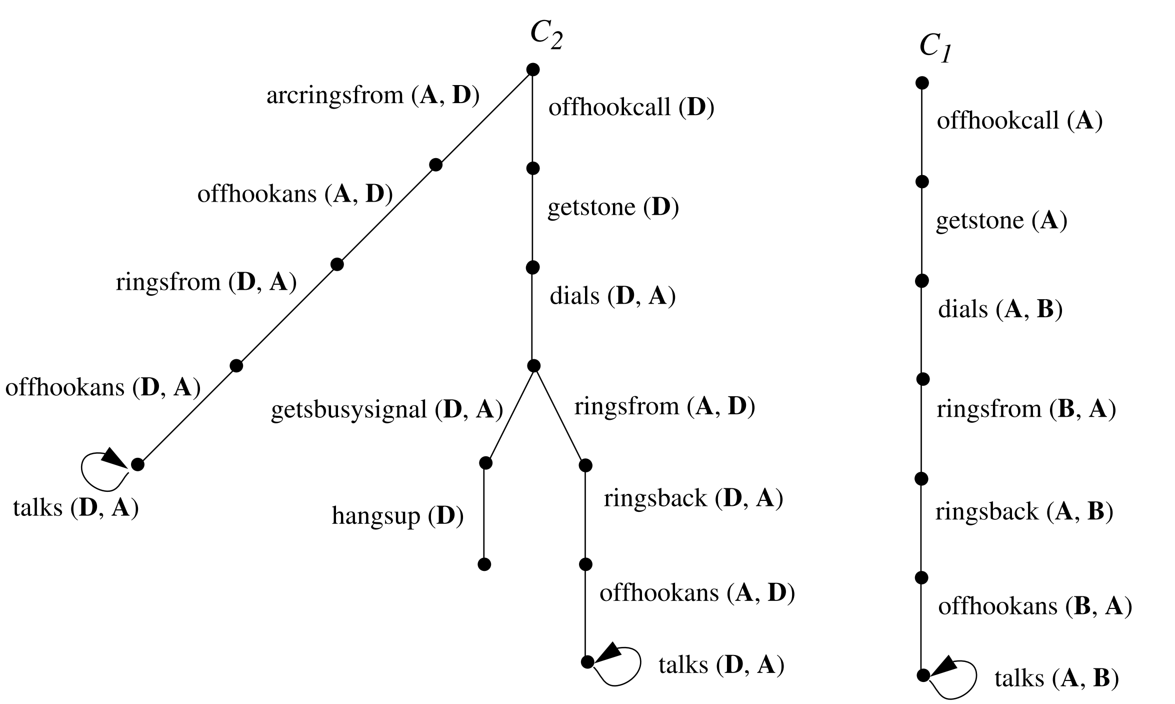


Fig. 6.10 LTS of *Arc* in the context of POTS.

with the sequence *offhookcall(**D**), getstone(**D**), dials(**D**, **A**),* then **D** would receive a busy signal followed by a hang up, after which it returns to the initial state. When **A** becomes idle, C_2 would attempt to establish a connection between **D** and **A**, by executing the action *arcringsfrom(**A**,**D**)*.

6.2.2.3 Informal Description of Tcs

*Terminating Call Screening (Tcs) is a feature which allows a subscriber to reject incoming calls from subscribers whose telephone numbers are included in the Tcs set. The subscriber is responsible for the creation and modification of the set. For example, in the figure below, user **A** would reject incoming calls from user **D**. Of course, **A** may still make calls to subscribers in his/her Tcs list.*

6.2.2.4 Formal Specification of Tcs

The structure of integrating *Tcs* is shown in Figure 6.11. Figure 6.12 shows the corresponding behaviour. In the figure, we assume that **A** has activated the *Tcs* feature so that incoming calls from **D** are refused. The types of signals or messages that **D** should receive are implementation details. In our specifications, these are represented by an abstract action, *refusecall*. Since *Tcs(**A**)* does not depend on the state of **A**, it is sufficient to explain the behaviour of C_3. So, when **D** calls **A**, **D**'s call is refused as expressed by the sequence *offhookcall(**D**), getstone(**D**), dials(**D**, **A**), refusecall(**D**, **A**)*. The behaviour of C_1 (not shown) is the same as POTS.

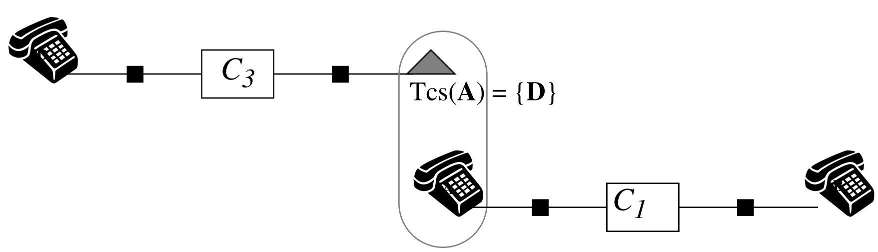


Fig. 6.11 Integrating *Tcs* into POTS

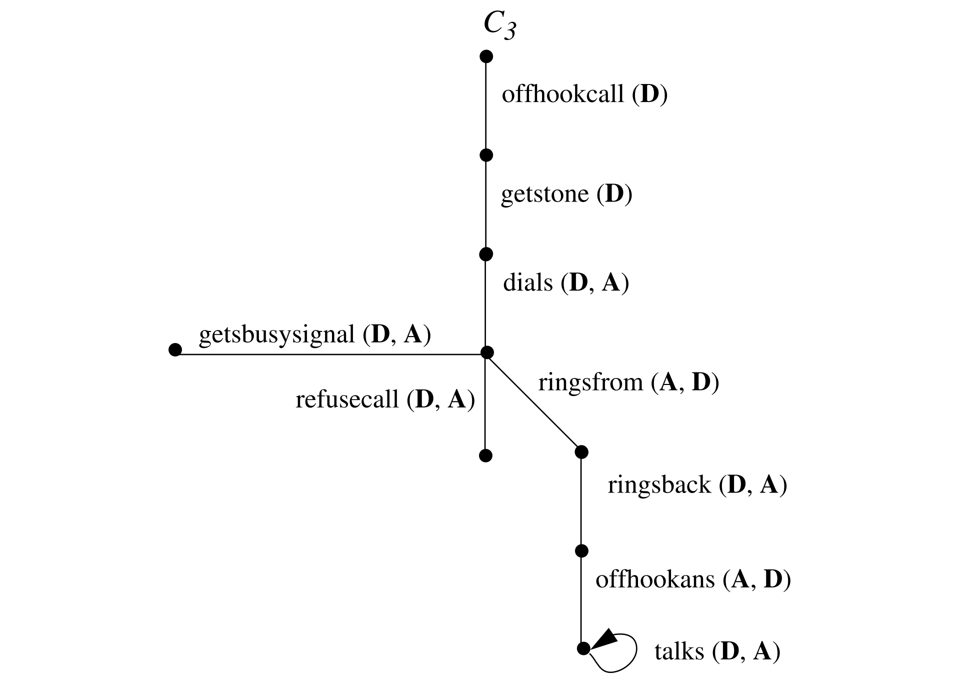


Fig. 6.12 LTS of *Tcs* in the context of POTS.

6.2.2.5 Integration of Tcs and Arc into POTS

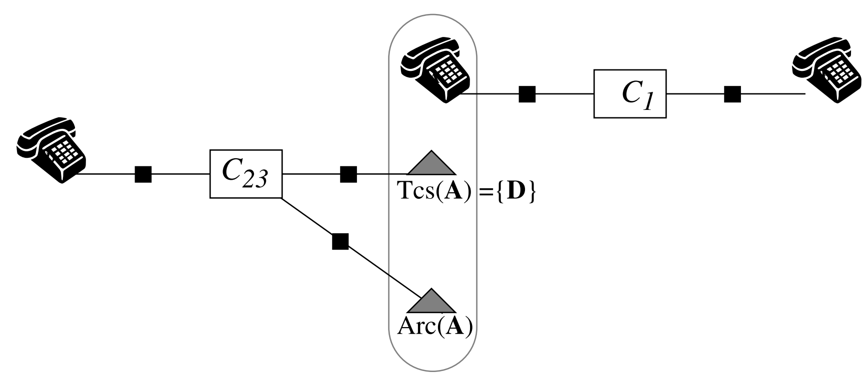


Fig. 6.13 Integrating *Tcs* and *Arc* into POTS

Figure 6.13 shows the resulting specification structure of integrating *Tcs* and *Arc* into POTS. The corresponding partial behaviour of the system, which we express by

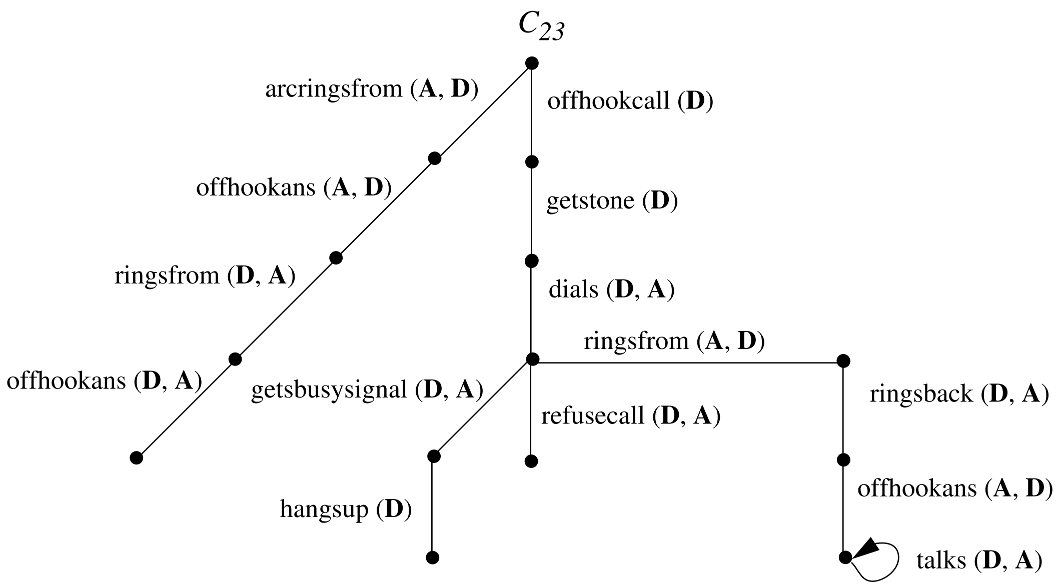


Fig. 6.14 LTS of the integration *POTS[Tcs*Arc]*.

*POTS[Tcs*Arc]*, is shown in Figure 6.14. This behaviour is based on the two LTSs, C_2 of Figure 6.10 and C_3 of Figure 6.12, respectively. Note that this figure reflects only a partial space of the system state, because we must consider the behaviour of C_1 as well as that of the global constraints, in order to form the complete system behaviour.

Let us verify that the functionalities of both features, when activated one at a time, are included in the behaviour of the integration. For *Tcs*, suppose that **D** is in the terminating call screening list of **A**. If **D** tries to call **A**, then two cases are possible:

(1) if **A** is busy, **D** receives a busy signal then hangs up to terminate the connection;

(2) if **A** is idle, **D**'s call is refused because it is included in **A**'s terminating screening list.

So, in either case, **D** cannot reach **A**. Now, let us check the functionality of *Arc*. Using the same scenario, two cases are possible as well. If **A** is idle, the communication becomes a normal POTS call. If **A** is busy, **A**'s number is retained. The execution follows the same sequence as in case (1) and C_{23} returns to the initial state, i.e., the root of the tree in Figure 6.14. When **A** becomes idle, another connection attempt is made, by executing the action *arcringsfrom (**A**, **D**)*.

The fact that both functionalities are independently preserved in the final specification does not guarantee that both features can be activated simultaneously, as justified in the next section.

6.2.2.6 Detection of Interactions between Tcs and Arc

Let us use a typical communication scenario to explain how the interaction occurs, given the two assumptions:(1) **A** has activated both features on its line, and (2) **A** is talking to **B**.

Now suppose that **D** attempts to call **A**. Since **A** is busy, it seems perfectly logical that a busy treatment be returned to **D**. Recall that if **A** is idle, **D** would have been prevented from reaching **A** because of the *Tcs(**A**)* feature. So, **D** is not able to reach **A** in either case. This is an interesting design decision that can easily be overlooked. What is the difference between returning a busy treatment and simply blocking the call when the called user (i.e. **A**) is busy? as it turns out, the result is the same if one considers the presence of *Tcs* by itself. However, the wrong decision introduces a side effect that manifests itself when one attempts to integrate another feature such as *Arc*, because the actions of the latter feature also depend on the busy state of **A**. So, in the presence of both features, *Arc* may simply register **D** as the user to which a call should be returned without screening it.

Figure 6.15 shows two simplified behaviours B_{arc} and B_{tcs} of the two features, as indicated by their subscripts. These two behaviours synchronize on the first three actions, because they are shared with POTS, and interleave on *getsbusysignal(**D**, **A**)* and *refusecall(**D**, **A**)*. When *Arc(**A**)* executes its action, the called user **A** is registered by the global constraint and a busy signal is returned to the caller **D**. When *Tcs(**A**)* executes its action, a *refusecall* signal is returned to the caller **D**.

A starting point for checking that these two features interact with each other is to execute their integration against the two testing processes t_1 and t_2, which are derived from the composition $POTS[B_{tcs}|[]|B_{arc}]$. It can verified that the integration *(I)* of Figure 6.14 on page 104 leads to a deadlock against both t_1 and t_2, from which we conclude that an interaction exists between *Tcs* and *Arc*. With respect to t_1, $I||t_1$ deadlocks on the action *hangsup(**D**)* which does not match the action *refusecall(**D**, **A**)*; with respect to t_2, $I||t_2$ deadlocks on the action *getsbusysignal(**D**, **A**)* which is offered by t_2 but not by *I*.

One way to solve the interaction between these two features is to give a priority to *Tcs* so that incoming calls are screened regardless of the state of **A**. This, of course, would prevent *Arc* from being activated if the caller is in the terminating screening list of **A**.

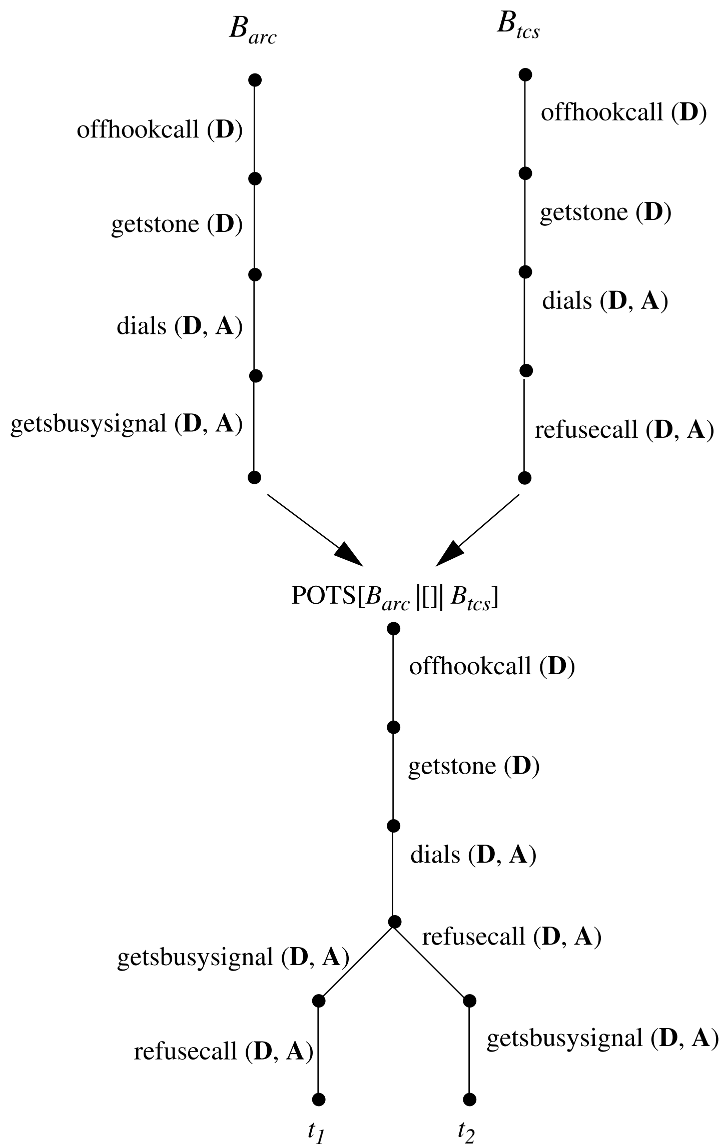


Fig. 6.15 Partial behaviour of the composition *POTS[B_{arc} |[]| B_{tcs}]*.

6.2.3 Example 3: *Call Forwarding Always and Originating Call Screening*

6.2.3.1 Informal Description of Ocs

Originating Call Screening (Ocs) is a feature which allows a subscriber to prevent outgoing calls to be made to a predefined set of numbers. The Ocs subscriber can still be reached from subscribers whose telephone numbers are included in the Ocs set. The subscriber is responsible for the creation and modification of the set. For example, in the figure below, user ***A*** *would be prevented from making outgoing calls to* ***B***. *Of course,* ***A*** *may still receive calls from* ***B***.

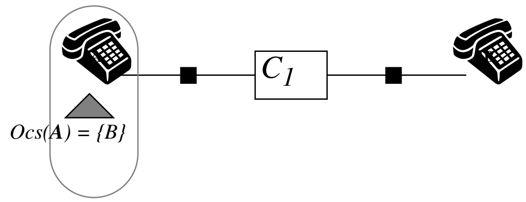


Fig. 6.16   Structure of the specification which integrates *Oc*s into POTS.

6.2.3.2 Formal Specification of Ocs

As can be seen from the structure of the specification in Figure 6.16, and the corresponding behaviour of Figure 6.17, *Ocs* is similar to *Tcs*, except that *Ocs* has a caller role whereas *Tcs* has a called role. *Ocs* can be expressed as a process which imposes further constraints on the caller side with respect to the POTS connection. One such constraint allows a called party **B** to ring for **A** only if **B** is not in the *Ocs* set of **A**. So, after the *dials(**A**, **B**)* action in the LTS, the system now offers a new alternative, the action *refusecall(**A**,**B**)*, to indicate that a connection from **A** to **B** is not possible if **B** is in the *Ocs* set. Note that the predicates, [**B** ∉ Ocs(**A**)] and [**B** ∈ Ocs(**A**)], which we associate with the actions *ringsfrom* and *refusecall* would normally be expressed in the global constraints process; we included them in this LTS to

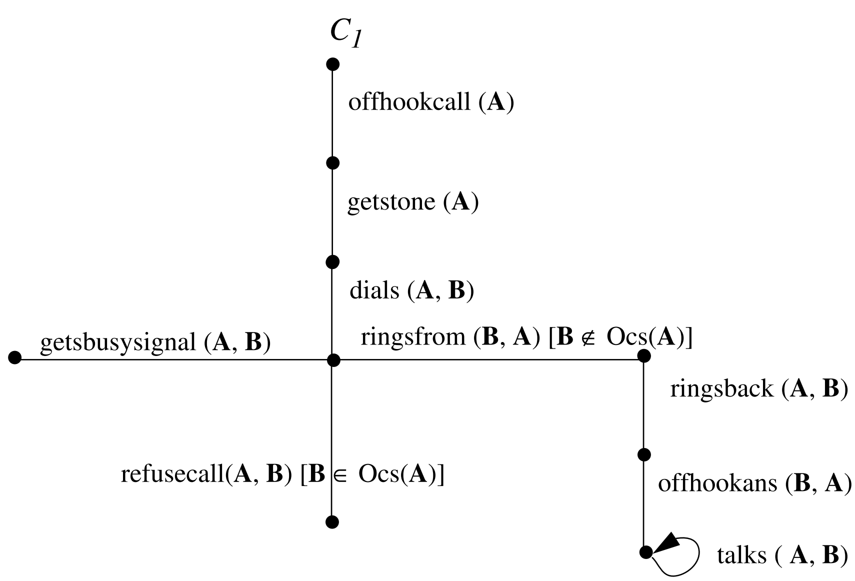


Fig. 6.17   LTS of *Ocs* in the context of POTS.

simplify the explanation. If **A** dials a user who is not in the set, then the system's behaviour is reduced to that of a normal POTS call.

6.2.3.3 Informal Description of Cfa

Call Forwarding Always (Cfa) has a similar functionality to that of Cfbl, except that it forwards all incoming calls to another preset telephone number, regardless of the state of the subscriber on which the feature is active. In the example below, ***A*** *always forwards his/her incoming calls to* ***D****.*

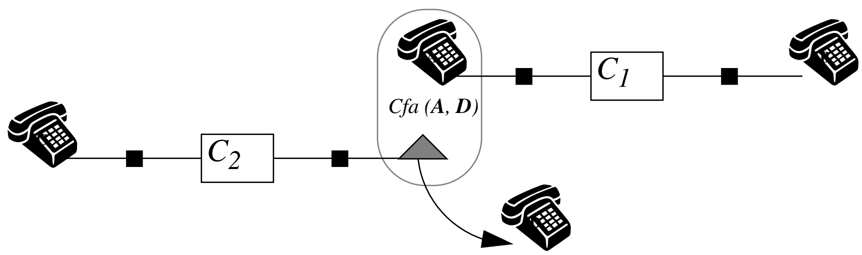


Fig. 6.18 Structure of the specification which integrates *Cfa* into POTS.

6.2.3.4 Formal Specification of Cfa

The structure of the specification which integrates *Cfa* in the context of POTS is shown in Figure 6.18. If we assume that **A** activates *Cfa(**A**, D)*, then a call from **C** to **A** is always forwarded to **D**, regardless of whether **A** is busy or idle. In general, **C** may call any user *y*. If *y* does not subscribe to *Cfa*, the call is reduced to a normal POTS connection between **C** and *y*. If *y* corresponds to a subscriber of *Cfa* such as **A**, the *Cfa (**A**, **D**)* is instantiated, and if **D** is idle, C_2 synchronizes with it on the *ringsfrom (**D**, **A**)* signal, as shown in Figure 6.19. After the connection is established between **C** and **D**, the call behaves like a basic POTS between the two users.

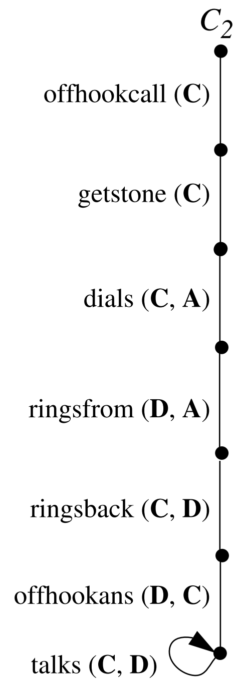


Fig. 6.19 LTS of *Cfa* in the context of POTS.

6.2.3.5 Integration of Cfa and Ocs into POTS

We start the integration by taking Figure 6.17, which integrates *Ocs* into POTS, as our starting point. We then extend it by adding the functionality of *Cfa*. The new structure is given in Figure 6.20. The corresponding behaviour is shown in Figure 6.21. The figure shows two processes, C_1 and C_2, which interleave with each other and fully synchronize with the global constraints process. In this example, we are mostly interested in the synchronization of the global constraints with each of C_1 and C_2, on the action *ringsfrom(y, x)* $[y \notin Ocs(x)]$, where *x* is the caller side and *y* is the called side.

Process C_1 synchronizes with the global constraints process on this action only if the associated guard is evaluated to true. By simple substitution of **B** for *x* and **A** for *y,* we can verify that $[\boldsymbol{B} \notin Ocs(\mathbf{A})]$ is evaluated to *false*. Therefore, no synchronization on this action occurs. However, C_1 does synchronize with the global constraints on the action *refusecall(**A**, **B**)*, which is what one would expect from the *Ocs* feature.

Process C_2 synchronizes with the global constraints process on the action *ringsfrom(y, x)*, because we do not consider the effect of the guard *[**D** $\notin$ Ocs(**A**)]*. Therefore, C_2 continues with the execution of its sequence, which leads to a connection between **C** and **D**, which is what we would expect from the *Cfa* feature.

This analysis justifies, informally, that the functionalities of each of these two features is supported by POTS when checked independently. In the next section, we show how to detect an interaction between them, when they execute simultaneously.

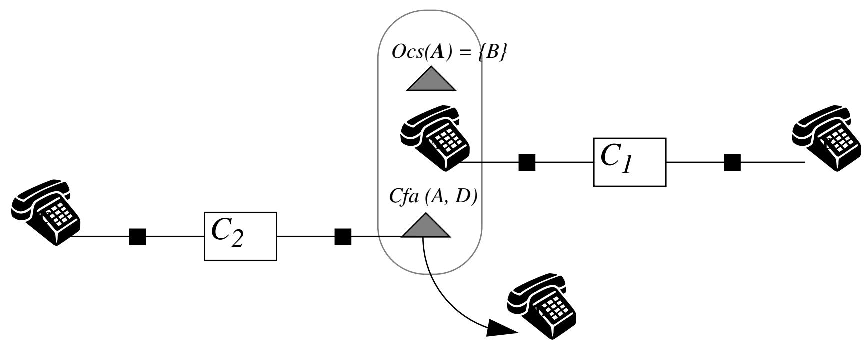


Fig. 6.20 Integrating *Cfa* and *Ocs* into POTS: Example 1

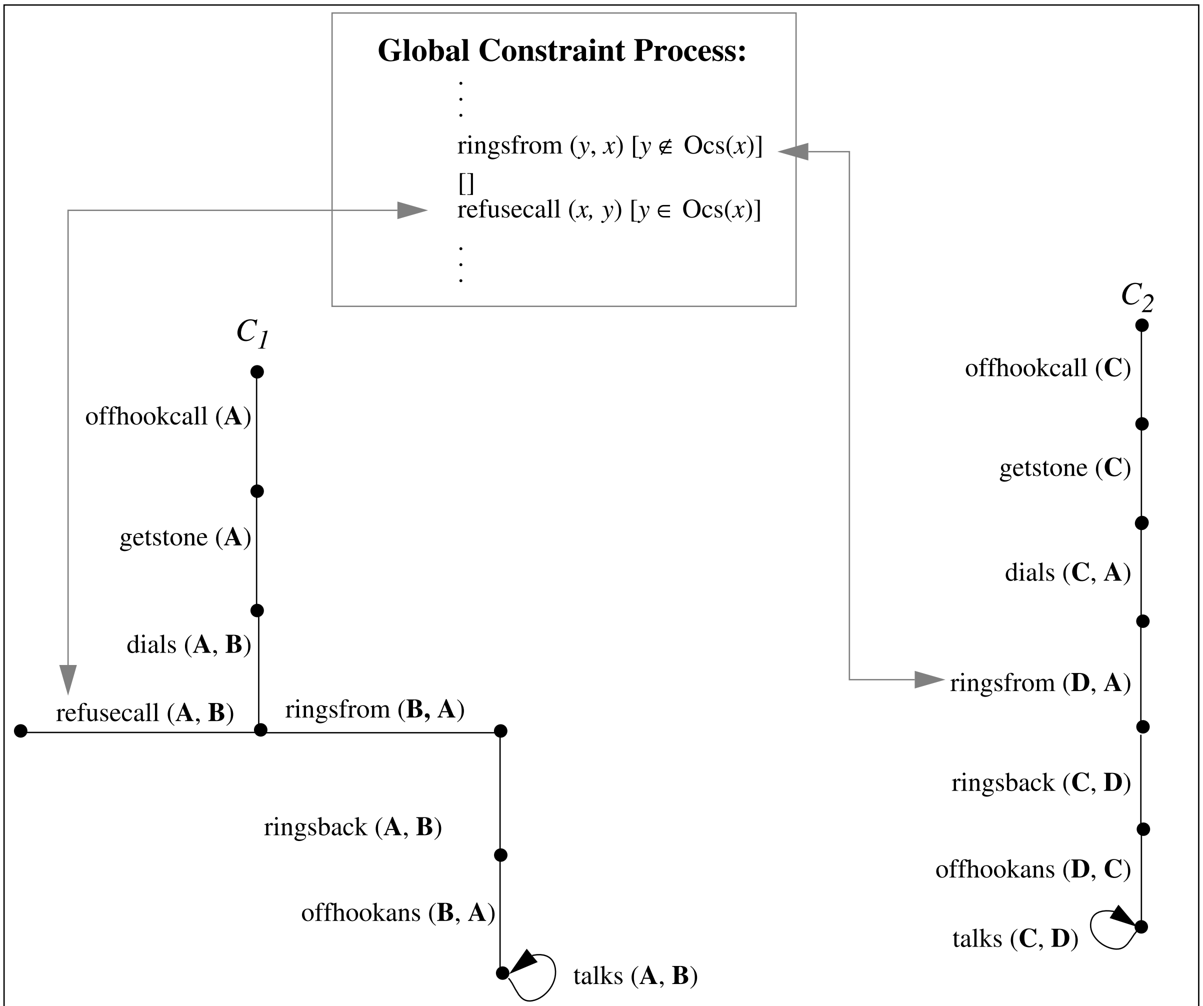


Fig. 6.21 LTS of *Ocs* and *Cfa* in the context of POTS.

6.2.3.6 Detection of Interactions between Cfa and Ocs

Let us use an execution scenario to explain the interaction that occurs between *Cfa* and *Ocs*, and show how to detect it. First, we need the composition of the two features in the context of POTS. Since we can assume that such a composition exists, then there exists two sequences, B_{ocs} and B_{cfa}, as shown in Figure 6.22, which could be derived from the composition, because each of the sequences is a valid trace in the context of its corresponding integration. Therefore, each of the sequences, if executed against their respective *integration*s, reaches its terminal state.

Let us trace the execution of each of the two sequences against the integration, with the

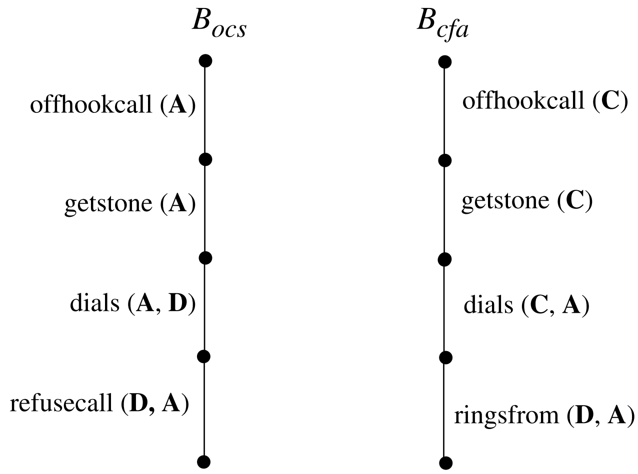


Fig. 6.22 Test Sequences for *Ocs* and *Cfa* (SUSE)

following assumptions, as shown in Figure 6.23.

(1) *Cfa* and *Ocs* are associated with **A**, and

(2) **A** calls **D** and **C** calls **A**.

Testing the system against B_{ocs} process does not lead to a deadlock as shown in Figure 6.24, which means that no conclusion can be drawn from the test. Testing the system against the process B_{cfa} leads to a deadlock as shown in Figure 6.25.

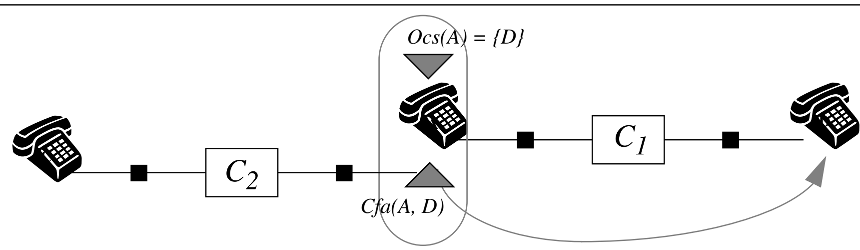


Fig. 6.23 Integrating of *Cfa* and *Ocs* into POTS: Example 2.

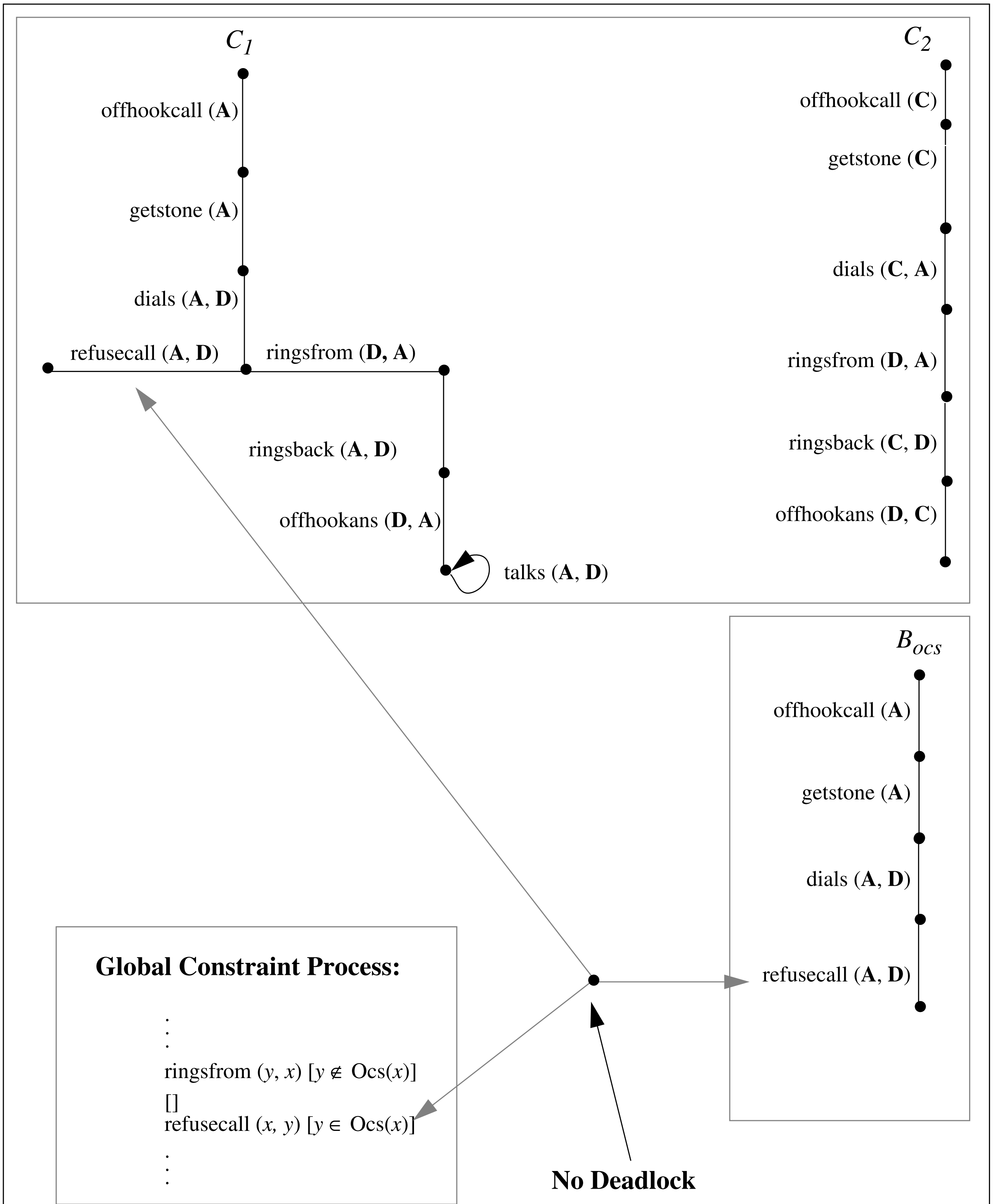


Fig. 6.24 Testing the integration against B_{ocs} testing process

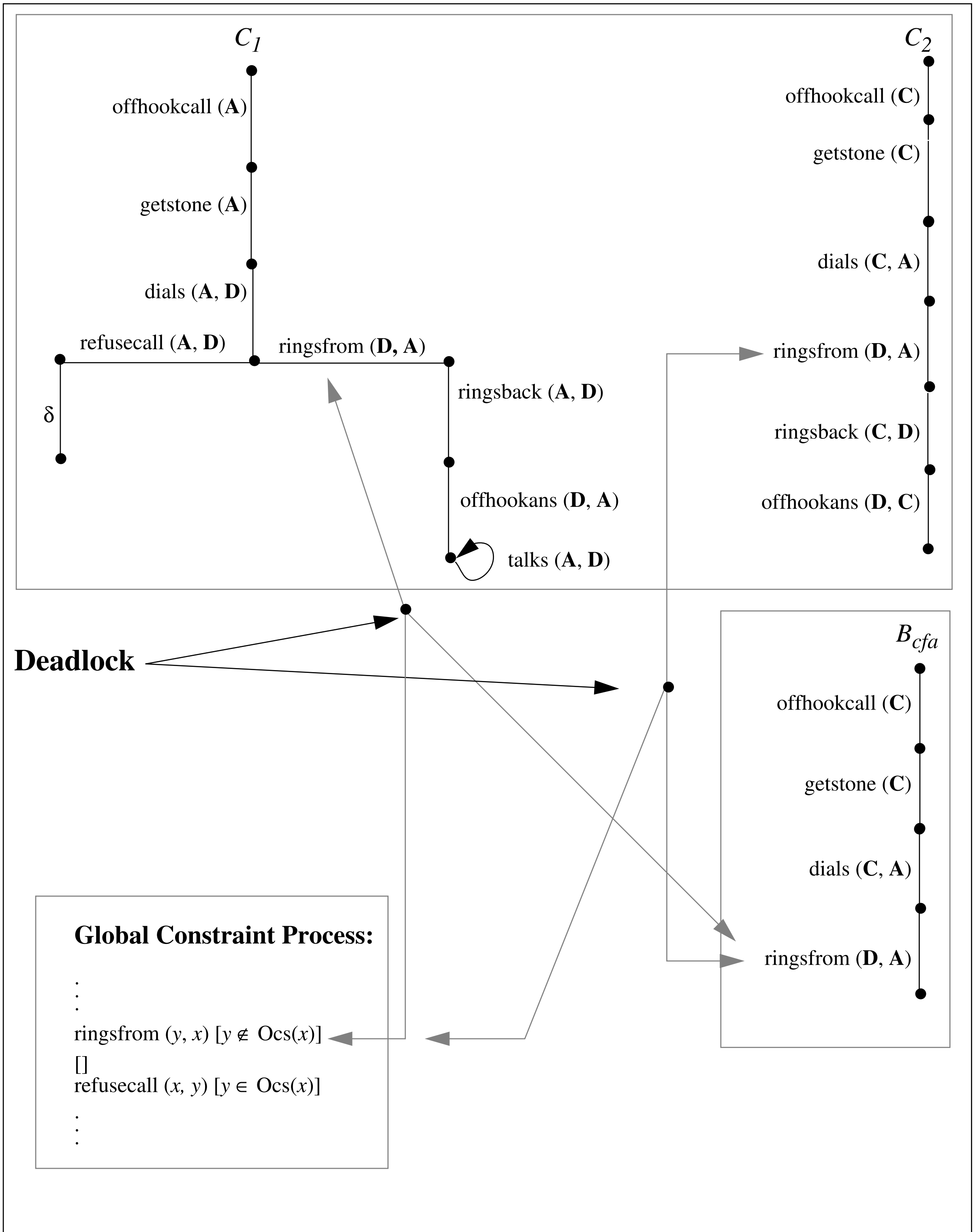


Fig. 6.25 Testing the integration against B_{cfa} testing process

6.2.4 **Example 4:** *Originating Call Screening and Distinctive Ringing*

The informal description and the formal specification of *Ocs* are given in Section 6.2.3 on page 106. Note that, starting from this example forward, we will deal with interactions of the MUSE types, meaning that the features will be associated with two different users.

6.2.4.1 Informal Description of Dsr

> *Distinctive ringing (Dsr) is a feature which allows a subscriber to distinguish between incoming calls, before answering them, based on their distinctive ringing patterns. Distinctive ringing has no effect on outgoing calls; they continue to be associated with the main number. Examples of this would be a household which acquires an additional number x to be activated on their main number* ***B****. The two numbers* ***B*** *and x would be distinguished by their different ringing patterns.*

6.2.4.2 Formal Specification of Dsr

Figure 6.26 shows the specification structure of the *Dsr* feature in the context of POTS. The feature acts on behalf of the called side of a POTS user. The behaviour of the resulting integration is shown in Figure 6.27. It assumes that the number *x is* associated with the main

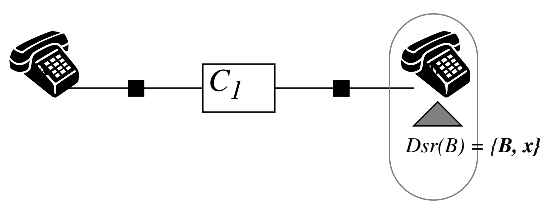


Fig. 6.26 Integrating *Dsr* into POTS.

number **B**. Such assumptions are part of the global constraints process. When a user such as **A** calls one of these two numbers, the ringing pattern at the called side distinguishes whom the call is intended for. This is indicated by the two different ringing branches, after the *Dials(**A***, y) action in the LTS, where *y* is either *x* or **B**. If the call is intended for **B**, the controller synchronizes with the POTS process, on the action *ringsfrom*(**B**, **A**). If the call is intended for *x*, the *Dsr(**B**)* process is activated, and synchronization occurs on the action *ringsfrom*(*x*, **A**). Actually, the POTS process and the *Dsr(**B**)* process are instances of the same definition; they differ only in their parameters, such as the intended called and the type of the ringing pattern. Also, since the global constraints process has knowledge of the relationship between these two numbers, a potential caller for either numbers would receive a busy signal, if either of the numbers is busy. This is indicated by the action *getsbusysignal*(**A**, *y*), where *y* is either **B** or *x*.

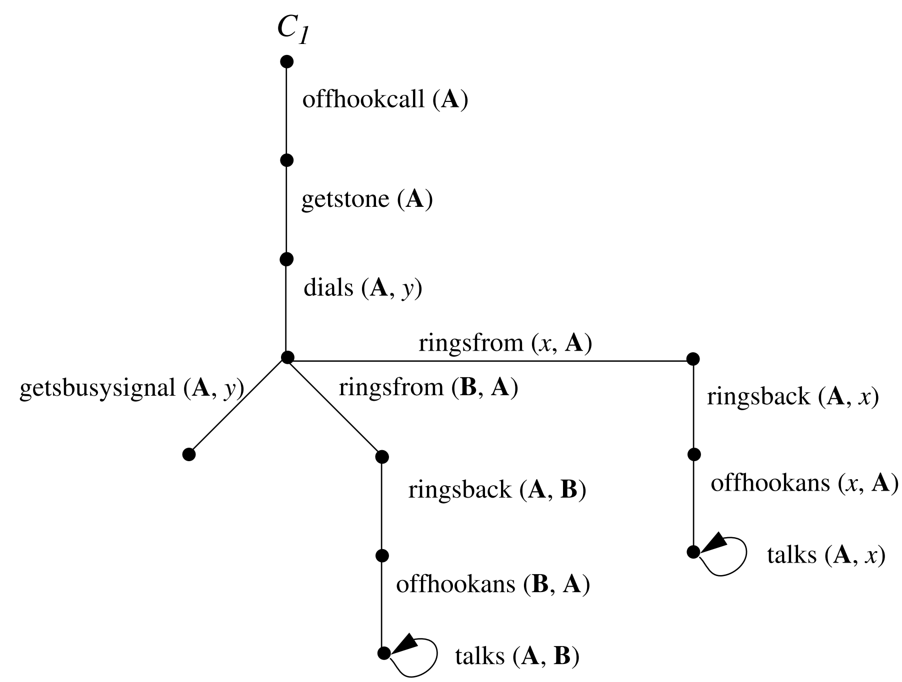


Fig. 6.27 LTS of *Dsr* in the context of POTS.

6.2.4.3 Integration of Ocs and Dsr into POTS

Following the same approach as in the previous examples, the specification's structure and behaviour of integrating the two features are shown in Figure 6.28 and Figure 6.29, respectively.

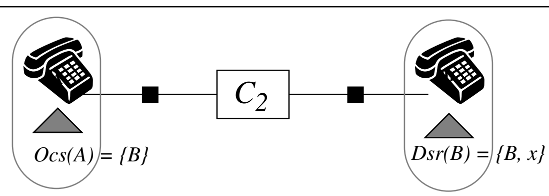


Fig. 6.28 Integration of *Dsr* and *Ocs* into POTS.

After the *dials(**A**, y)* action there are five possible actions, in the global state, from which a selection can be made. Note that actions (1) and (2) below are represented by a single action in the figure, where *y* is either **B** or *x*:

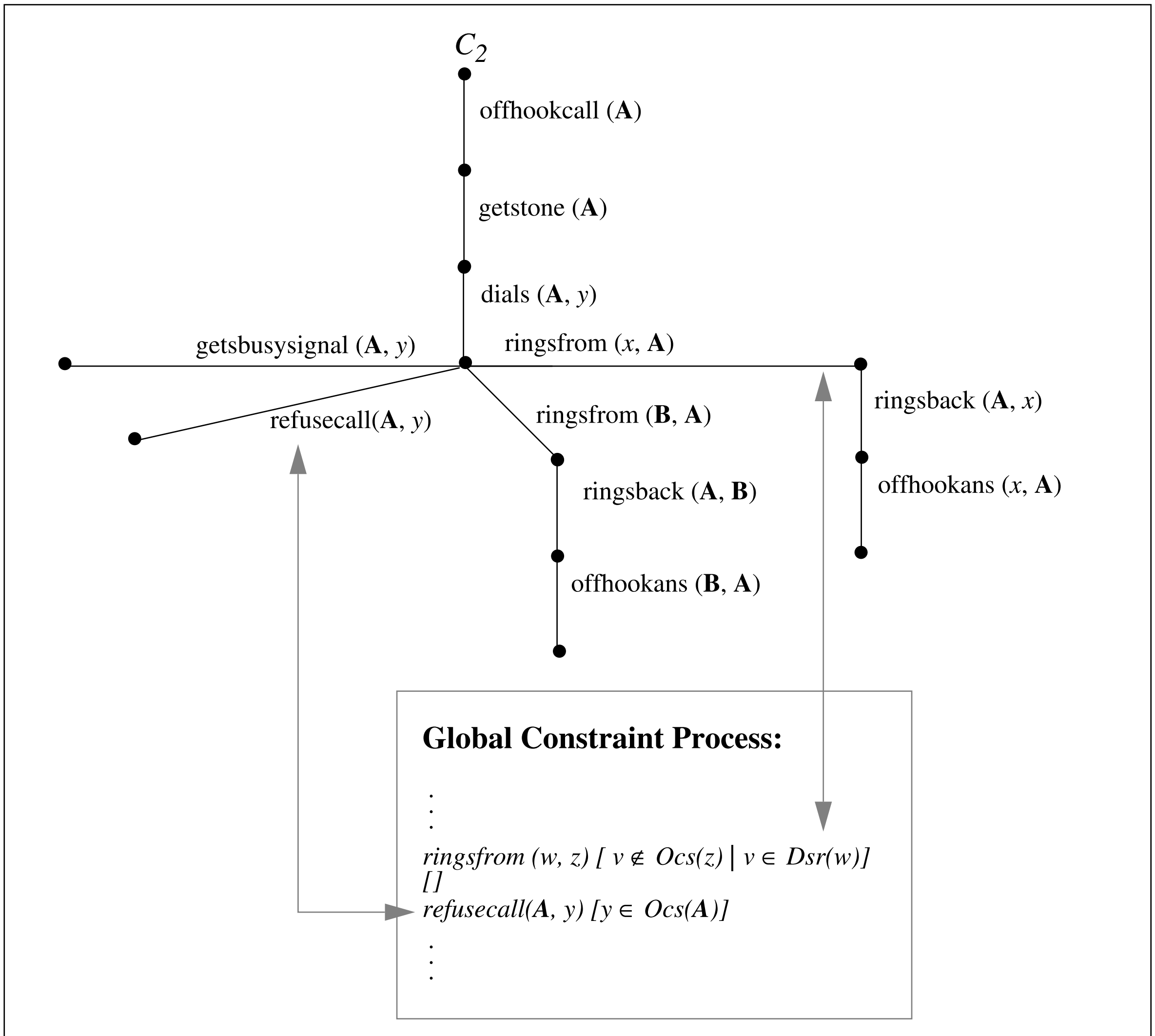


Fig. 6.29 LTS of of the integration *POTS[Dsr*Ocs]*

(1) *getsbusysignal* (**A**, **B**): if **A** tries to call **B** (i.e., *y* = **B)**, and either **B** or *x* is busy;

(2) *getsbusysignal* (**A**, *x*): if **A** tries to call *x* (i.e., *y* = *x)*, and either **B** or *x* is busy;

(3) *refusecall(**A**, y)*: The call from **A** to **B** is blocked if *y* is in *Ocs(**A**)*;

(4) *ringsfrom* (**B**, **A**): the POTS process representing the main number synchronizes with the controller using the POTS ringing pattern.

(5) *ringsfrom* (*x*, ***A***): if **A** calls *x*, the *Dsr(**B**, x)* takes control of the execution and synchronizes with the controller using *x's* distinctive ringing pattern.

The guard *[z ∉ Ocs(**A**) | z ∈ Dsr(y)]*, which expresses a global constraint on the execution of the action, allows us to verify that none of the users associated with the distinctive ringing list of the called user *x* are in the originating call screening list of the caller. Otherwise, the call is blocked as expressed by the guard associated with the *refusecall(**A**, y)* signal, expressed by the global constraint as well.

In the next section, we will analyse this integration to show how to detect an interaction between these two features.

6.2.4.4 Detection of Interactions between Ocs and Dsr

Figure 6.30 shows a valid sequence of actions with respect to *Ocs*, when considered on its own. Since all traces which are included in the integration of the feature with respect to

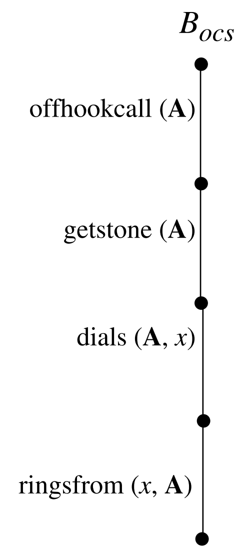


Fig. 6.30 Test sequence for *Ocs* and *Dsr* (MUSE)

POTS are also included in the *composition* of the two features, then the trace shown above can be considered as a test sequence. Figure 6.31 shows the results of the execution, where a deadlock occurs on an action that passed the test when executing the *Ocs* feature independently. Assume that **B**, who subscribes to a *Dsr* feature with a secondary number *x*, is idle, and that **A**, who has included **B** in his/her originating screening list, attempts to call *x*. Then, after the *dials(**A**, y)* action, where *y* is *x*, a deadlock occurs because none of the five possible actions synchronize with the tester.

(1) *getsbusysignal* (**A**, **B**): since **B** is idle, then this action is not possible;

(2) *refusecall(**A**, **B**)*: since the intended called is *x* and not **B**, then this action is not

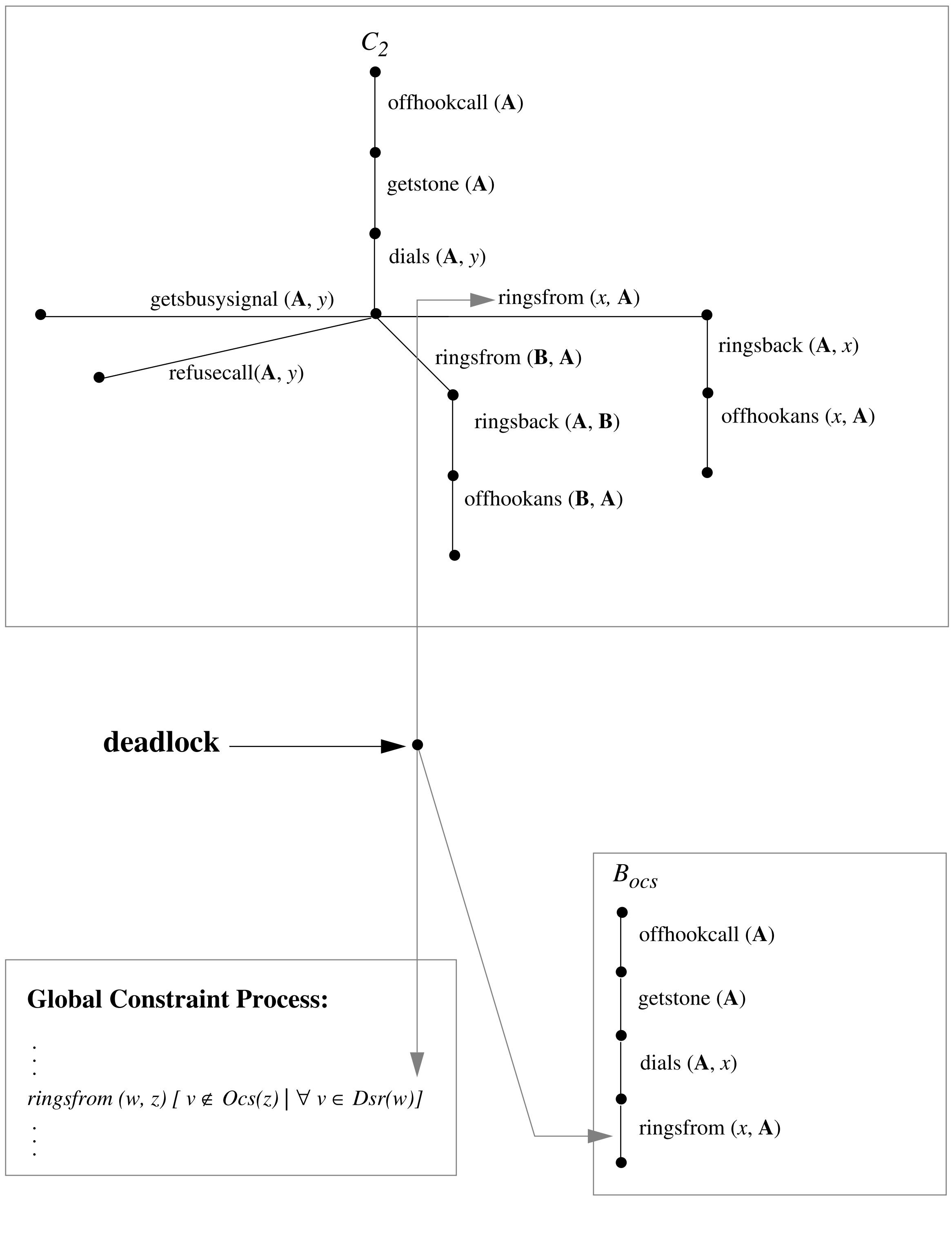


Fig. 6.31 Testing the integration against B_{ocs} testing process

possible either;

(3) the same holds for the action *ringsfrom* (***B***, ***A***); finally,

(4) *ringsfrom* (*x*, ***A***) is not possible either, because the guard $[v \notin Ocs(z) \mid \forall v \in Dsr(w)]$ in the global constraint is evaluated to *false*, since $[\mathbf{B} \notin Ocs(\mathbf{A})] \wedge [\mathbf{x} \notin Ocs(\mathbf{A})]$ is *false*.

Therefore, a trace which is valid in the context of *Ocs* by itself results in a deadlock when the two features are executed simultaneously, which indicates the existence of an interaction.

The simplest solution to avoid the interaction between these two features is to disable one of them while the other one is active. Another solution would be to use a dynamic negotiation model, as described in [GrVe92], to resolve the interaction at run time.

6.2.5 Example 5: *Call Forwarding Always and Originating Call Screening - Revisited -*

These two features were introduced in Section 6.2.3 on page 106. They were analysed for interactions with respect to a single user.

6.2.5.1 Integration of Cfa and Ocs into POTS

In this example, we consider the case where the features are associated with two different users, **A** and **B**. The structure of the specification which integrates the two features in the context of POTS is shown in Figure 6.32. The corresponding integration *(I) POTS[Cfa*Ocs]* is shown in Figure 6.33. In this integration, we consider a system with three

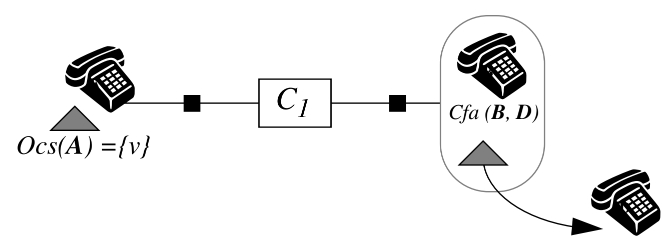


Fig. 6.32 Integrating Cfa and *Ocs* into POTS - MUSE.

users **A**, **B**, and **D**.

Before we show how to detect the interaction between these two features, let us argue informally that both features exhibit their behaviours when considered one at a time. For *Ocs,* if we assume that *Ocs*(***A***)={***B***} and **A** calls **B**, then the following execution shows the refusal of the connection between **A** and **B**: *offhookcall(**A**); getstone(**A**); dials(**A**,**B**); refusecall(**A**,**B**)*. This sequence synchronizes with the integration on the first three actions because they are common between *Ocs* and POTS. It also synchronizes on the fourth action, *refusecall (**A**, **B**)*, because the guard $[z \in Ocs(y)]$ in the global constraints process is evaluated to *true* when both x and z are instantiated with **B** and y is instantiated with **A**. Similarly, *offhookcall(**A**); getstone(**A**); dials(**A**,**B**); ringsfrom(**B**,**A**)* is a valid sequence if **A** calls **B** and $v \neq \boldsymbol{B}$.

For *Cfa*, if we assume that **A** calls **B**, and **B** forwards all incoming calls to **D**, then the following execution: *offhookcall(**A**); getstone(**A**); dials (**A**,**B**); ringsfrom(**D**,**A**)* shows that the call is forwarded to **D**. The same reasoning holds for the first three actions. Since we do not take into account the effects of the conditions which are imposed by *Ocs*, and since *Cfa* imposes no constraints that prevent the fourth action, *ringsfrom(**D**,**A**)*, from being executed, synchronization occurs on this action as well.

6.2.5.2 Detection of Interactions between Cfa and Ocs

Assume that *Ocs(**A**) = {**D**}* and *Cfa(**B**, D)*. Then, **A** cannot call **D** directly, but it seems perfectly legal to call **B**, whose calls are always forwarded to **D**. Considering the execution of both features at the same time means that when **A** calls **B**, **D** should ring for **A**, as specified by *Cfa(**B**, **D**)*, and **A** should refuse the call, as specified by *Ocs(**A**)*. Formally, these two behaviours are expressed by the sequences B_{cfa} and B_{ocs}. These two sequences are used to obtain a partial composition, $POTS[B_{cfa}|[]|B_{ocs}]$, from which two testing processes are derived. Figure 6.34 shows the composition and the two tests, t_1 and t_2.

Allowing both features to execute simultaneously implies that the integration passes both t_1 and t_2. Figure 6.35 shows that a deadlock occurs when t_1 is used, which reveals an interaction between the two features. Here is how the deadlock occurs. The first three actions of t_1 synchronize with the integration because they are common between *Ocs* and POTS; however the system $I||t_1$ deadlocks on the fourth action, *ringsfrom (**D**, **A**)*, because the guard $[y \notin Ocs(z)]$ is evaluated to *false* when z and y are instantiated by **A** and **D**, respectively. A similar result is obtained by using t_2.

Again, an easy way to resolve this interaction is to disable one of the features when the second one active.

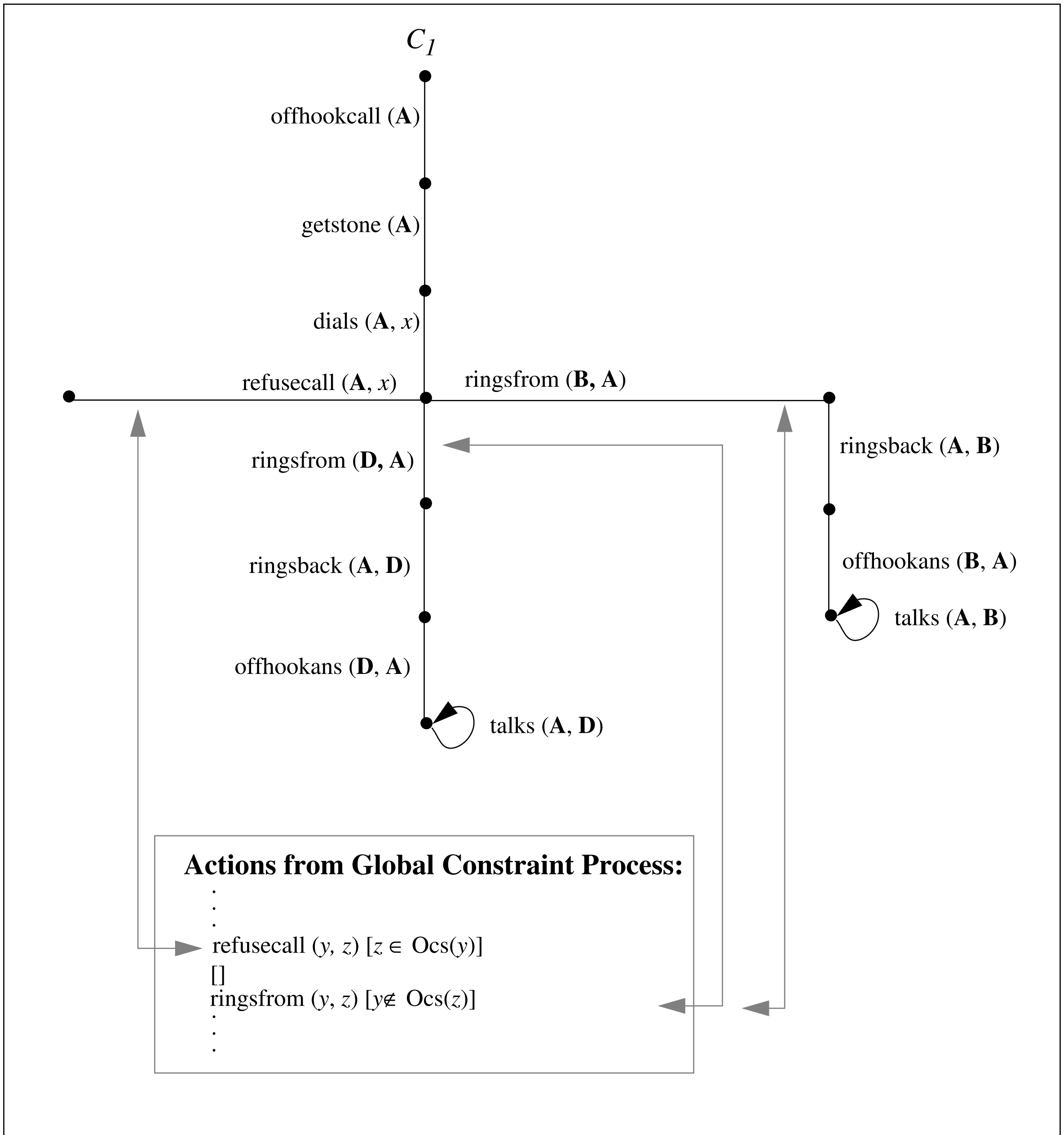


Fig. 6.33 LTS of *Ocs* and *Cfa* in the context of POTS - MUSE.

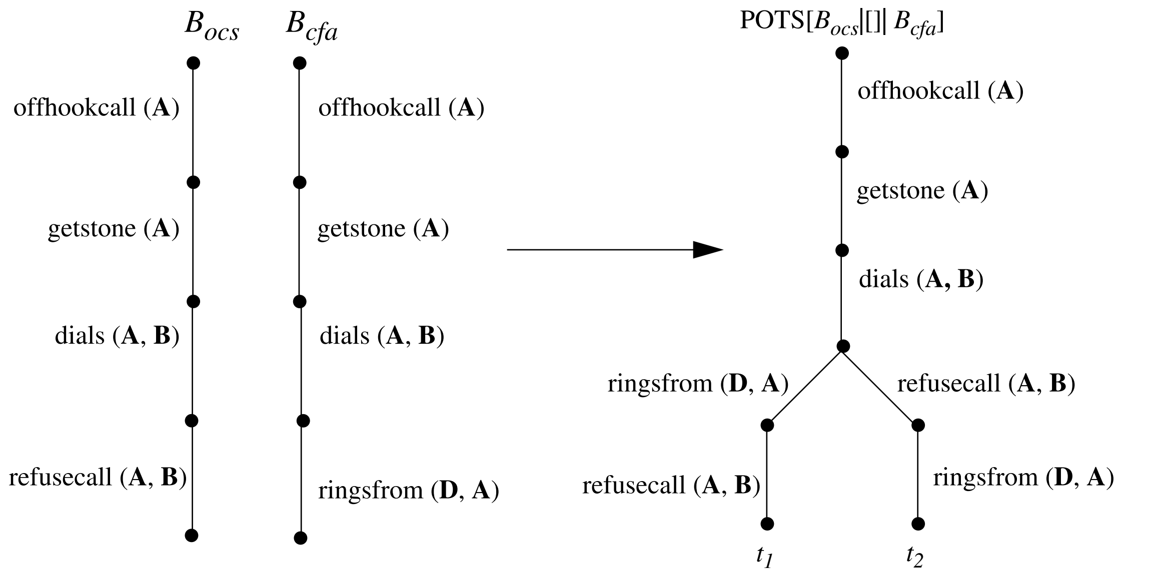


Fig. 6.34 Composition *POTS[Ocs|[]|Cfa]* and its test cases- MUSE.

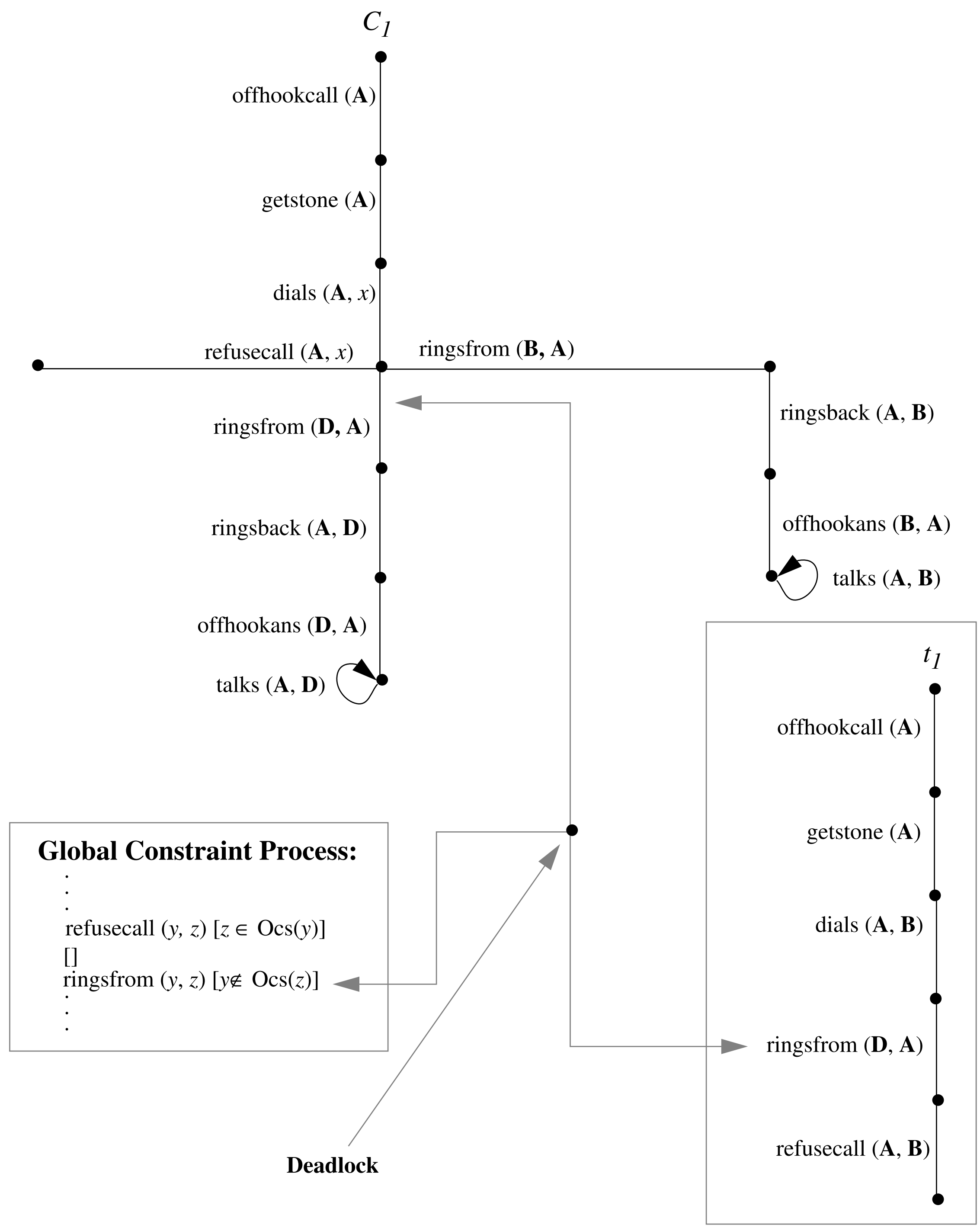


Fig. 6.35 Detection of interaction between *Ocs* and *Cfa* - MUSE-.

6.2.6 **Example 6:** *Call Waiting and Automatic CallBack*

See Section 6.2.1 on page 94 for the informal and formal description of *Cw*.

6.2.6.1 Informal Description of Acb

Acb works as follows. After a user, who subscribes to Acb, encounters a busy signal, he/she hangs up and activates the Acb feature. When the busy station goes on-hook and the calling station is on-hook, the calling station receives a distinctive ringing pattern. When answered, the call is automatically completed to the previously busy station.

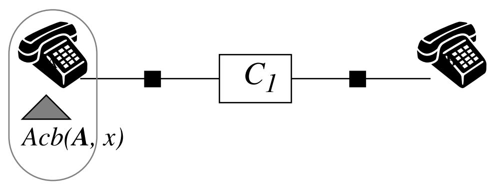


Fig. 6.36 Integration of *Cw* and *Acb* into POTS.

6.2.6.2 Formal Specification of Acb

Figure 6.37 shows the behaviour of integrating the *Acb*(**A**, *x*) feature in the context of POTS. We assume that **A** is a caller and **B** is a called. The LTS expresses that, after the

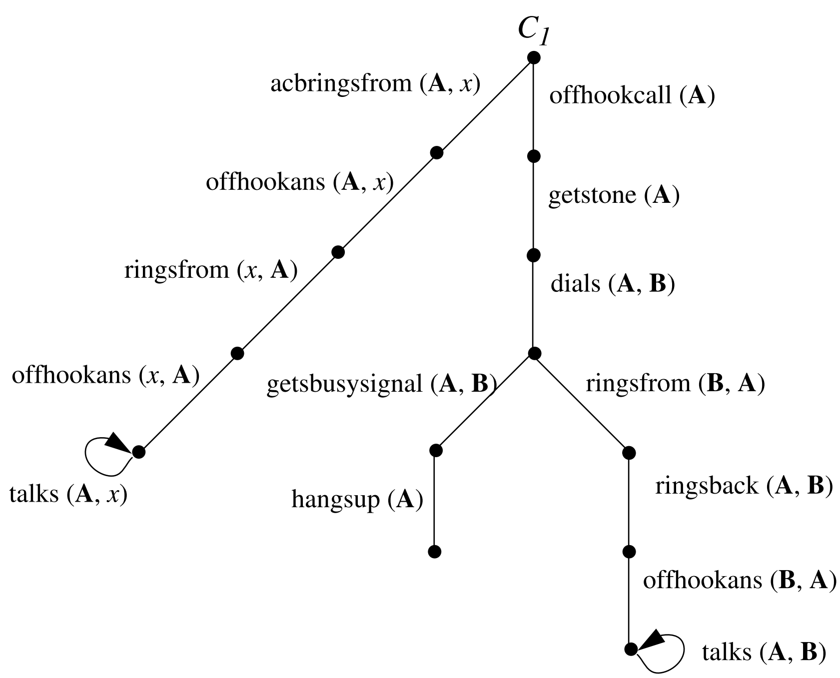


Fig. 6.37 LTS of *Acb* in the context of POTS.

*dials(**A**,**B**)* action, if **B** is idle, a normal POTS connection is established between **A** and **B**; if **B** is busy, **A** receives a busy signal, hangs up, and C_1 returns to the initial state. There is also an implicit assumption that the called station, **B** in this case, is registered by the global constraints and passed to the process *Acb*(**A**, *x*), as shown at the initial state of C_1. From this state, **A** can either initiate a POTS call or answer the distinctive ringing pattern related to *Acb*, an indication that **B** has become idle.

6.2.6.3 Integration of Cw and Acb into POTS

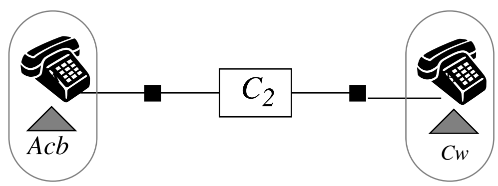


Fig. 6.38 Integration of *Cw* and *Acb* into POTS.

Figure 6.39 shows the behaviour of *POTS[Cw*Acb],* the integration *(I)* of *Cw* and *Acb* in the context of POTS. In the figure, we assume that user **A** has activated *Acb* and user **B** has activated *Cw*. Suppose that **B** is talking to **C** when **A** attempts to initiate a connection to **B**, then according to *Cw,* **B** would receive a call waiting tone and **A** would receive a normal ring back. If **B** answers the call, a talking session is established between **A** and **B**; if **B** does not answer, then **A** hangs up and **B** continues talking to **C**. According to *Acb*, since **B** is talking to **C**, then **A** receives a busy signal then hangs up, and the global constraints process registers **A** as the most recent called user. These two sequences, expressed by B_{acb} and B_{cw}, are shown in Figure 6.40.

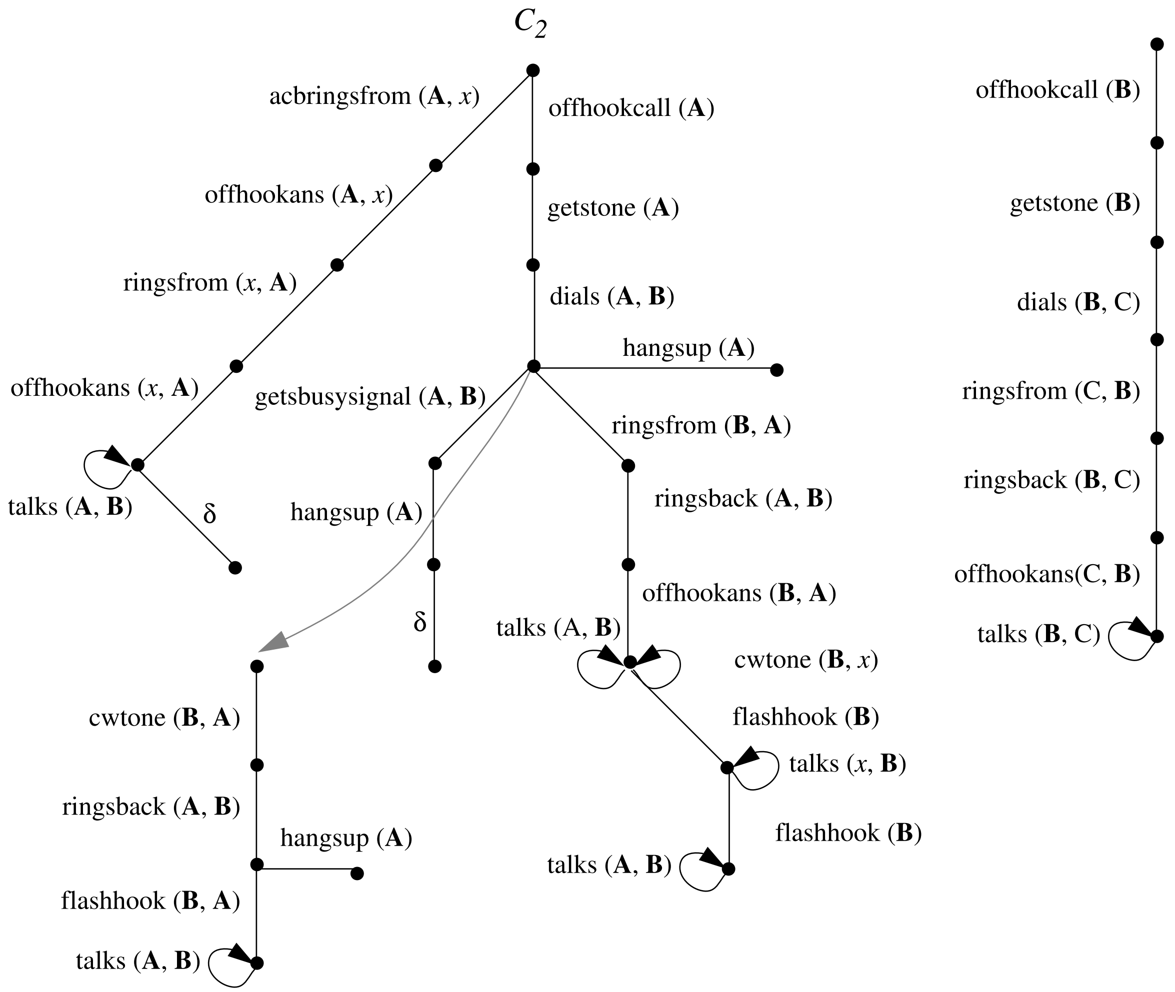


Fig. 6.39 LTS of *POTS[Cw* Acb]* - MUSE-

In the next section, we will show how to detect interactions between the two features.

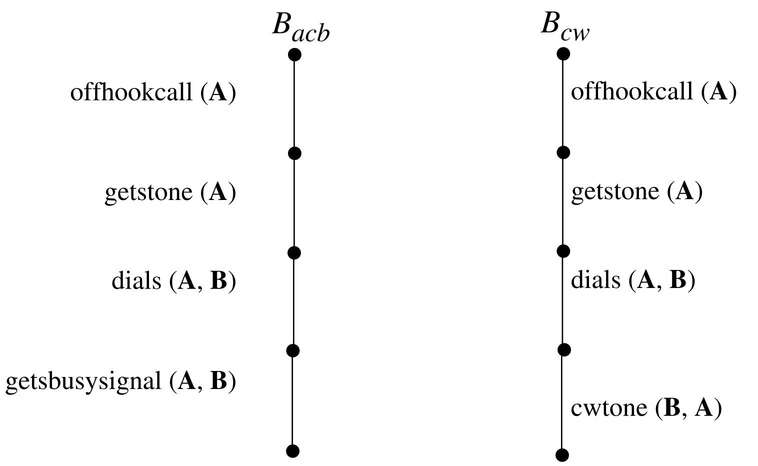


Fig. 6.40 Partial behaviours of *Acb* and *Cw* in the context of POTS.- MUSE -

6.2.6.4 Detection of Interactions between Cw and Acb

Figure 6.41 shows a partial composition, $POTS[B_{cw}\ |[]|B_{acb}]$, of *Cw* and *Acb* in the context of POTS. From this composition, we are able to generate two testing processes, t_1 and t_2, such that I deadlocks with both of them. The system $I||t_1$ deadlocks on the action *getsbusysignal (**A, B**)*, since the integration assumes that when an incoming call arrives, the *Cw*

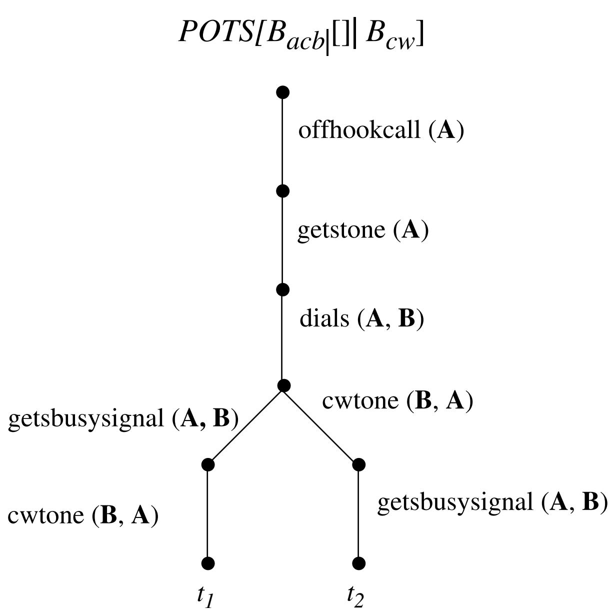


Fig. 6.41 Composition of *POTS[Acb*|[]| *Cw*] - MUSE -

feature overrides the *getsbusysignal (**A**, **B**)* action by executing the action *cwtone(**B**, **A**)*. This shows that *Cw* prevents *Acb* from executing one of its valid sequences, when considered in the context of POTS. We also detect a deadlock by executing the system I$\|t_2$. The deadlock occurs on the *getsbusysignal (**A**, **B**)* as well, since the integration does not offer this signal once the *cwtone* signal is executed.

In real life, this may not require a technical solution. It may be sufficient to inform the *Acb* subscriber of the actual behaviour of this feature in the context of other features such as *Cw*.

6.2.7 **Example 7:** *Call Waiting and Call Waiting*

Call Waiting was introduced in Section 6.2.1 on page 94.

6.2.7.1 Integration of Cw and Cw into POTS

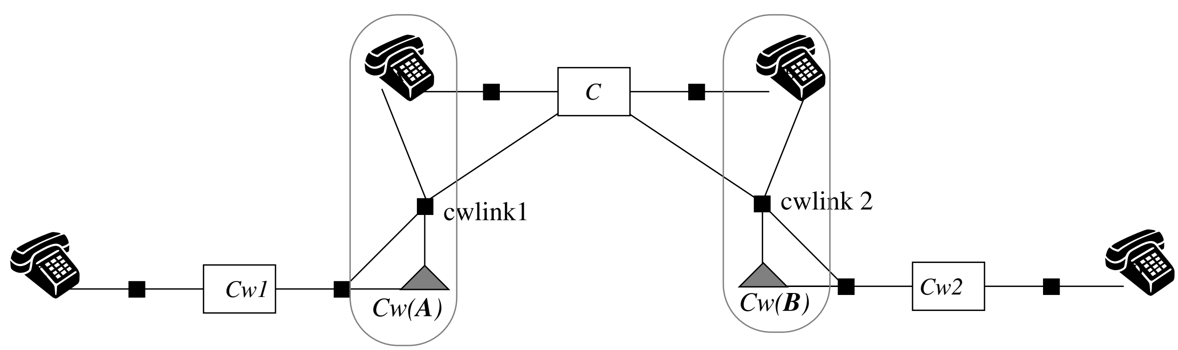


Fig. 6.42 Integrating Cw an Cw into POTS

Figure 6.42 shows the integration of a *Cw* feature to both a caller side and a called side of a POTS specification. In this figure, we assume that **A** is a caller and **B** is a called. The integration requires two modifications to the POTS controller which handles the connection between **A** and **B**. The first modification takes into account the integration of *Cw* into POTS so that the caller side (i.e. **A**) is able to respond, while talking to **B**, to another user such as **C**. The second modification of the POTS controller integrates the behaviour of the call waiting feature as seen from the called side (i.e., **B**). A partial global behaviour of the system can be expressed by the LTSs of Figure 6.43, which corresponds to the behaviours of *C*, *Cw1*, and *Cw2*. If only one user is allowed to activate his/her feature at a time, then each feature would behave as

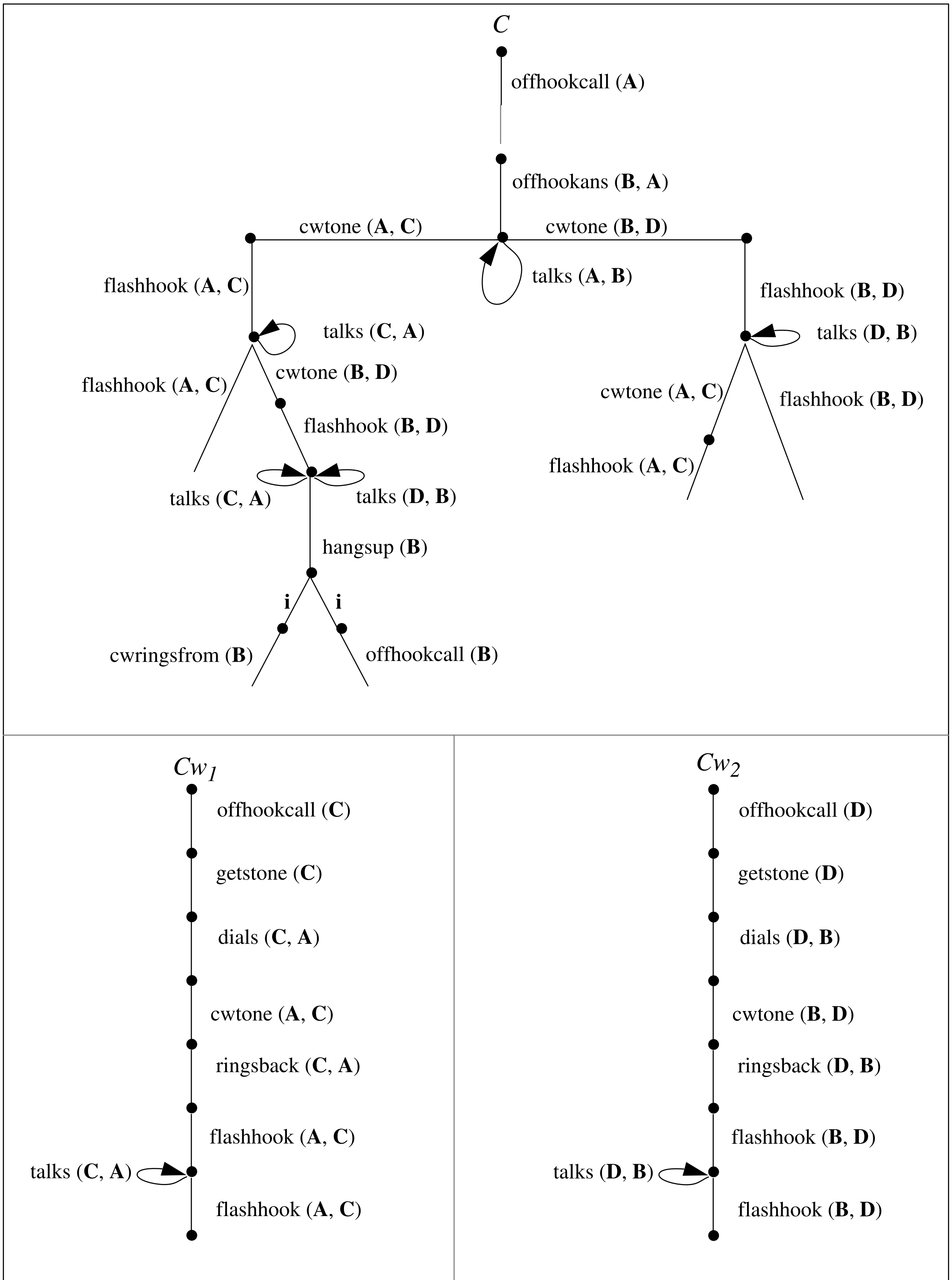


Fig. 6.43 LTSs of *Cw* and *Cw* in the context POTS

specified in the context of POTS. However, we show in the next section that allowing both **A** and **B** to activate their respective instances, at the same time, leads to an interaction between the two instances of *Cw*.

6.2.7.2 Detection of Interactions between Cw and Cw

Let us explain how the interaction occurs when both instances of *Cw* are used simultaneously. We assume that **A** has established a connection with **B**. Now suppose that **C** calls **A**, as specified by controller Cw_1. Then **A** may respond to **C** by flashing the hook, which

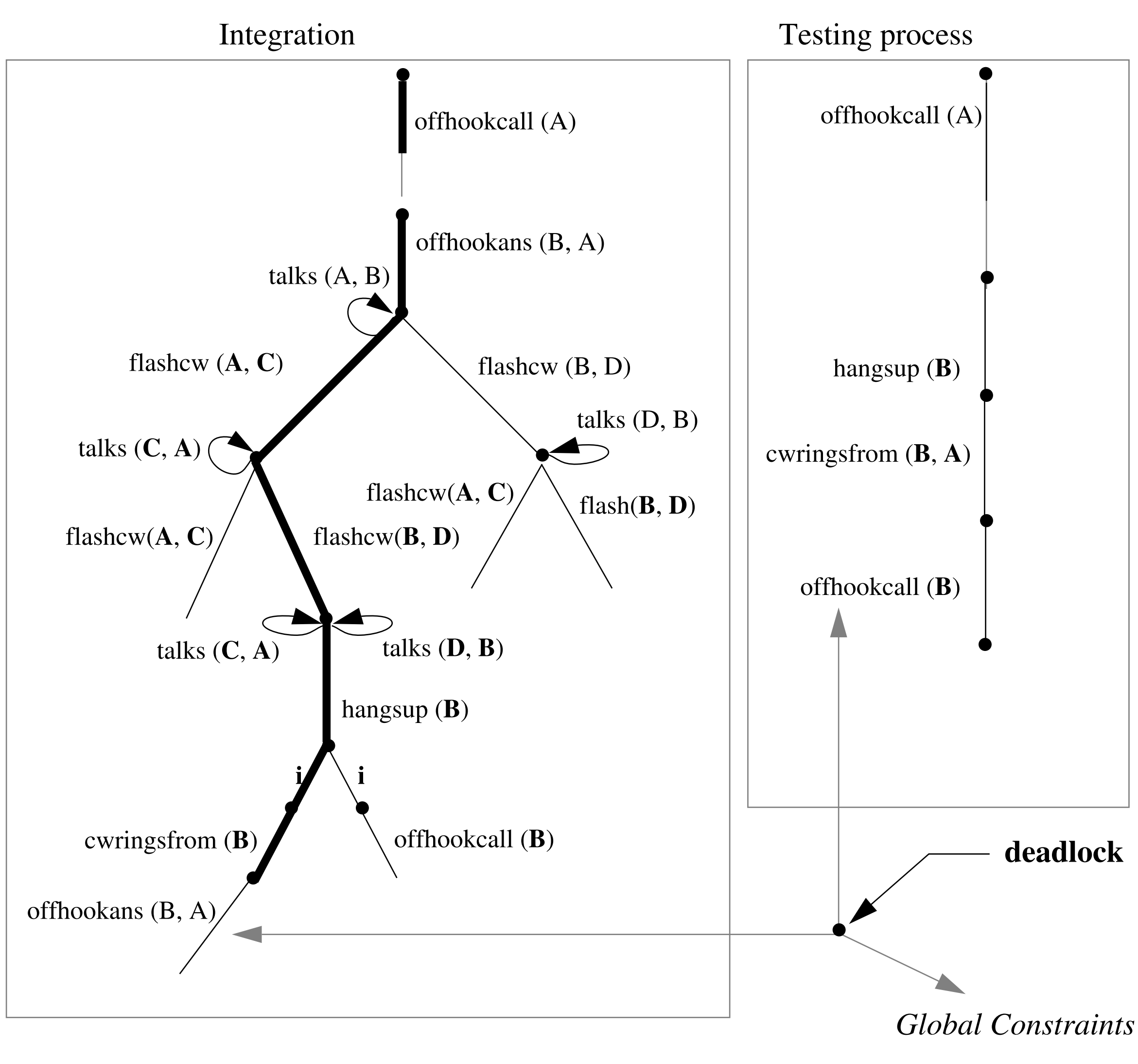


Fig. 6.44 Testing process for *Cw* and *Cw*.

puts **B** on hold and starts a talking session with **C**. Furthermore, suppose that **D** calls **B** according to controller Cw_2. Then **B**, who is in a held state, may flash the hook, which also puts **A** on hold and start a talking session with **D**. The system is now in a global state where **C** is talking to **A**, **D** is talking to **B**, **B** is being held by **A**, and **A** is being held by **B**. Suppose that **B** hangs up. Then, according to the *Cw*(**A**), a normal two way connection continues between **C** and **A**, whereas **B** returns to the initial state, from which new calls can be initiated or received. However, according to *Cw*(**B**), **B** must be rung back because **A** is still on hold. Therefore, there are two non-deterministic transition from the global state after the hang up. This is modelled by two internal actions in the integration. The testing process shown in Figure 6.44 detects such an interaction. If the integration selects the ***i*** on the left branch, the tester synchronizes with the integration up to, and including the action *cwringsfrom(**B**, **A**)*, after which **B** becomes busy, operating in the context of *Cw(**B**)*. Since **B** is now busy, then it may not initiate a new connection, as would have been possible in the context of *Cw*(**A**). Therefore, the system deadlocks on the next action *offhookcall(**B**)* of the tester, which doe not match the *offhookans(**B**, **A**)* of the integration.

6.2.8 Example 8: *Calling Number Delivery and Unlisted Number*

6.2.8.1 Informal Description of Cnd

> *Calling Number Delivery (Cnd) is a call-processing feature which delivers the caller's number to the called party's telephone during the ringing cycle.*

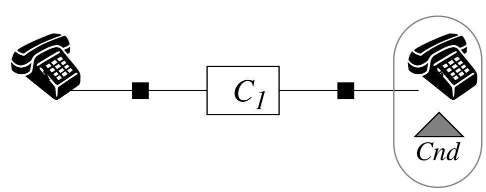


Fig. 6.45 Integration of *Cnd* into POTS.

6.2.8.2 Formal Specification of Cnd

The Formal specification of *Cnd* in the context of POTS is shown in Figure 6.46. Its behaviour deviates from that of POTS after the *dials* action. After the *dials* action is executed, the *ringsfrom* action is executed by the process of the feature instead of the POTS process. For

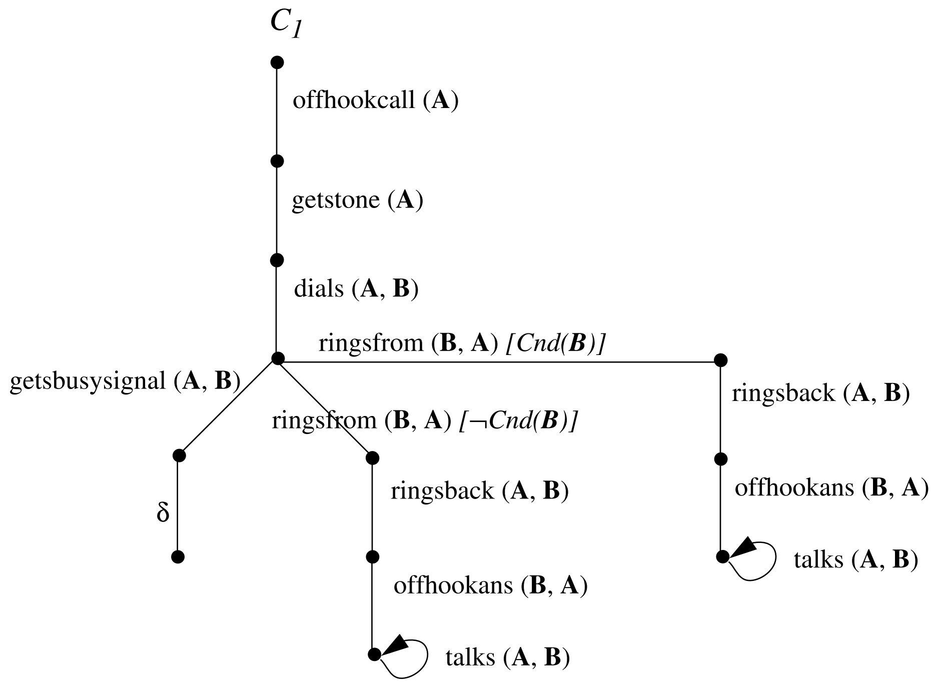


Fig. 6.46 LTS of *Cnd* in the context of POTS.

the rest of the behaviour, the process of the feature is similar to the called side of the POTS process. In the LTS below, the right branch represents the behaviour of the feature; the middle one represents the called side of POTS.

6.2.8.3 Informal Description of Uln

Unlisted Number (Uln) gives subscribers the option of keeping their telephone numbers private, so that the network cannot deliver the subscribers number to the called party during the ringing cycle.

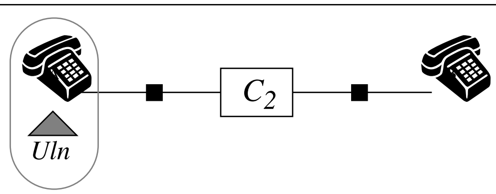


Fig. 6.47 Integration of *Uln* into POTS

6.2.8.4 Formal Specification of Uln

the specification of *Uln* is very similar to that of *Cnd*. Its functionality, on the other hand, is exactly the opposite of *Cnd*. It prevents the caller's number from being delivered, during the ring cycle, to the called side. Except for the guards, which in this case are imposed by the caller side, the LTSs of the two features are similar. Figure 6.48 shows the LTS of *Uln*.

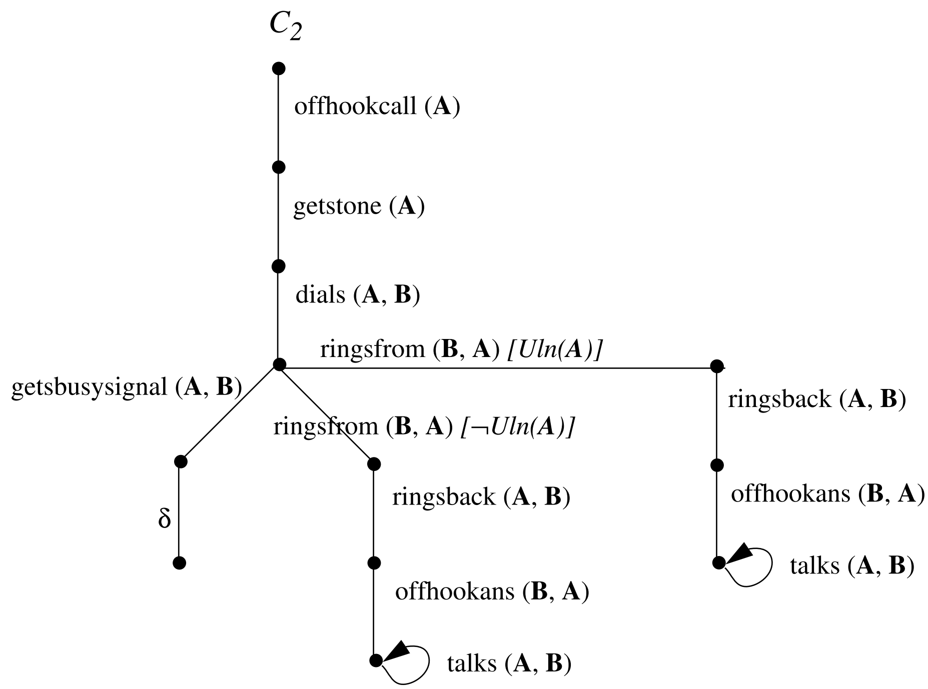


Fig. 6.48 LTS of *Uln* in the context of POTS.

6.2.8.5 Integration of Cnd and Uln into POTS and Detection of their Interactions

The integration of *Cnd* and *Uln* is shown in Figure 6.49. Its LTS behaviour is shown in Figure 6.52. The figure shows the three possible actions, after the *dials* action, which are of interest to our analysis.

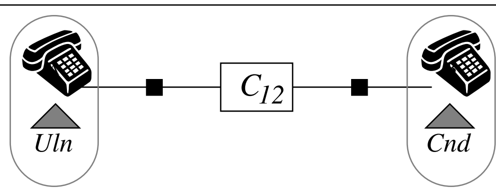


Fig. 6.49 Integration of *Cnd* and *Uln* into POTS

The question that needs to be answered is *can we allow both **A** and **B** to use their features at the same time?*, the answer is *no, we cannot.* The interaction between the two features is detected using a testing process, which we have derived as follows. Figure 6.50 shows two sequences, B_{cnd} and B_{uln}, both of which are valid with respect to their respective features. Therefore, all the testing sequences which are generated from the composition of

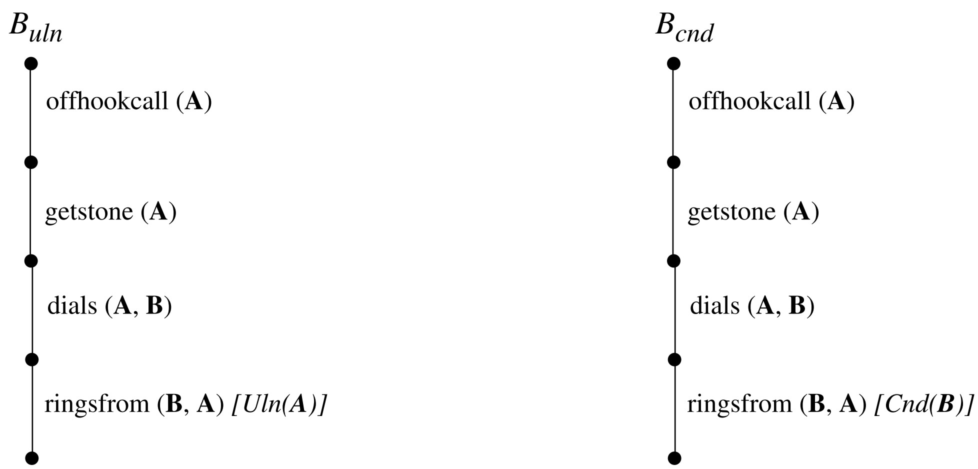


Fig. 6.50 Valid Traces for the Features *Uln* and *Cnd*.

POTS$[B_{cnd}|[\,]|t_{uln}]$ must be valid with respect to the integration, if the two features are allowed to operate simultaneously. However, the integration deadlocks against both t_1 and t_2. Figure 6.53 shows an execution using test case t_2, which indicates that an interaction exists with respect to this trace.

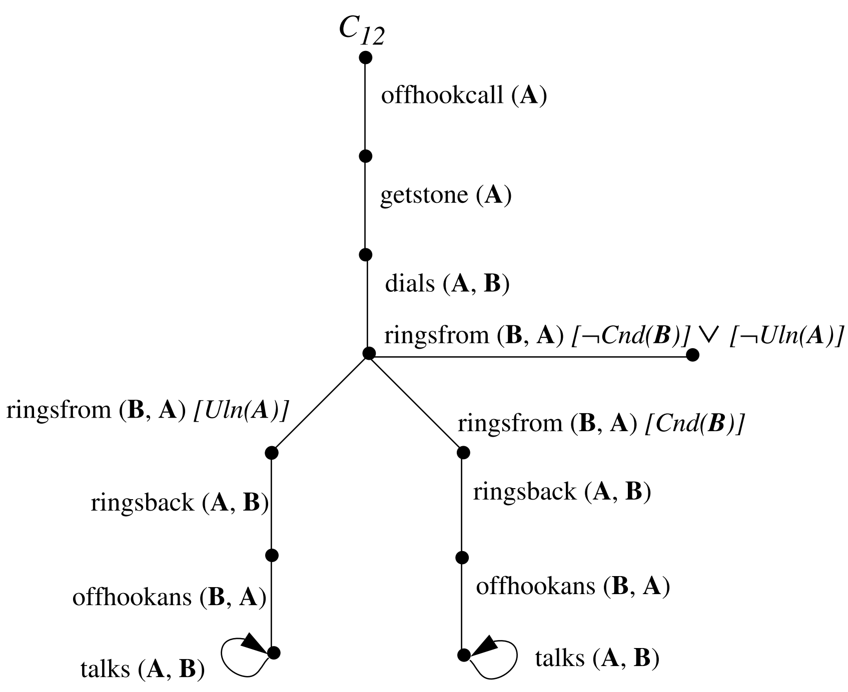


Fig. 6.51 LTS of *Uln* and *Cnd* in the context of POTS.

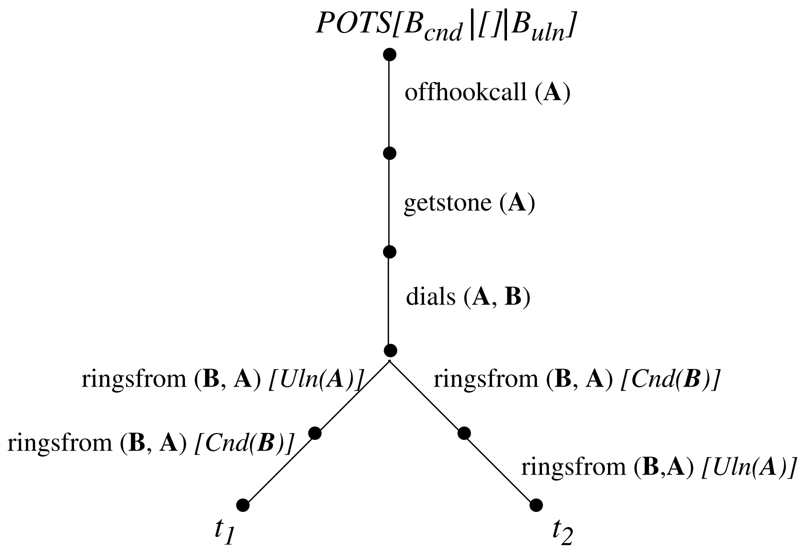


Fig. 6.52 Composition of *POTS[B_{uln}|[]|B_{cnd}]*

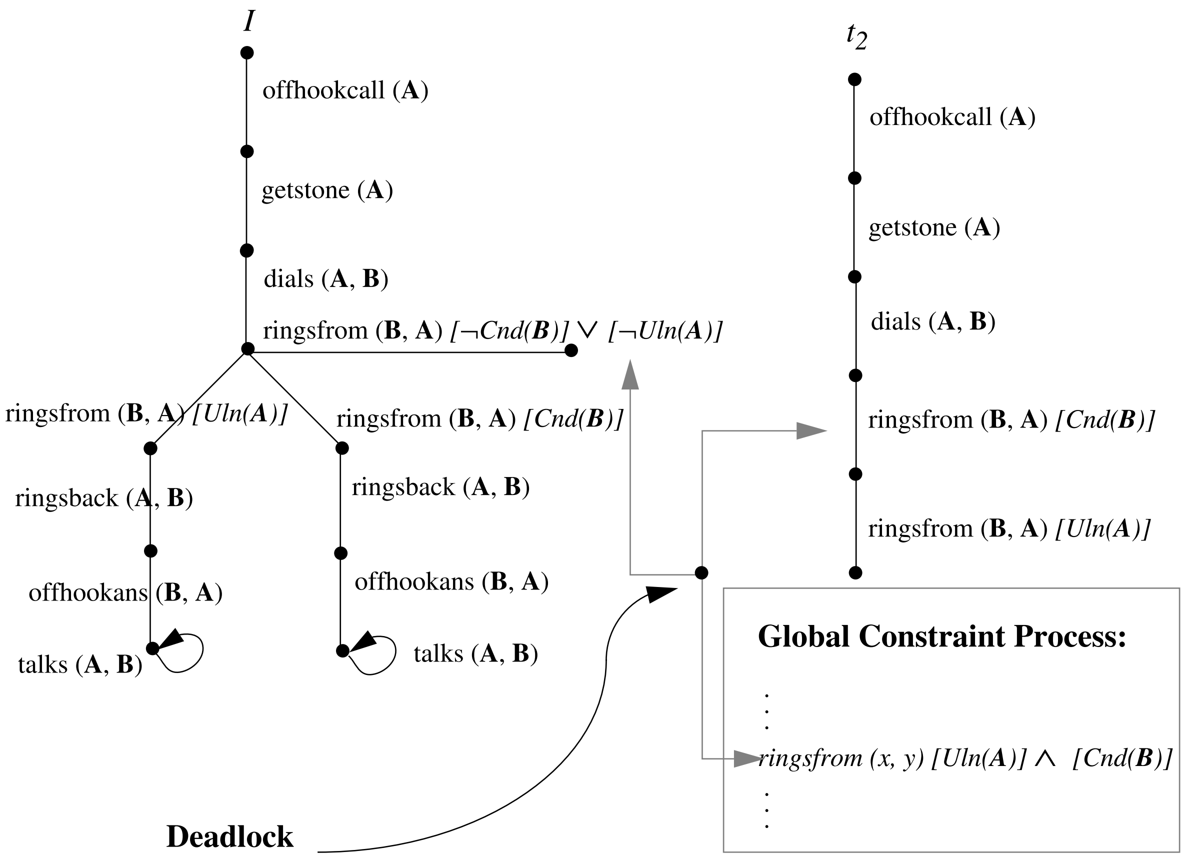


Fig. 6.53 Detection of an Interaction between of *Uln* and *Cnd*

6.2.9 **Example 9:** *Automatic CallBack and Automatic ReCall*

Acb was introduced in Section 6.2.2 on page 100; *Arc* was introduced in Section 6.2.6 on page 125.

6.2.9.1 Integration of Acb and Arc into POTS - MUSE

We illustrate the functionality of these two features in the context of POTS using a static

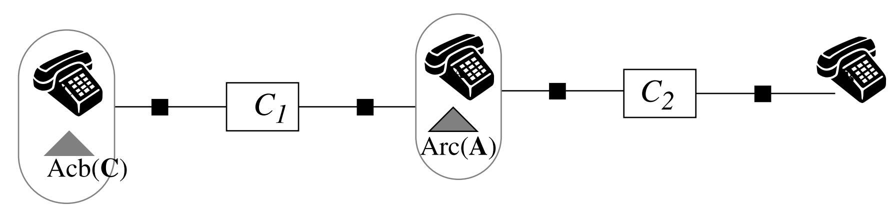


Fig. 6.54 Integrating *Acb* and *Arc* into POTS - MUSE

structure for three users **A**, **B**, and **C**, as shown in Figure 6.54. User **A** subscribes to *Arc* and **C** subscribes to *Acb*. The global behaviour of the system is composed, in addition to the global constraints process, of all the processes shown in the figure. The behaviour of C_1, the process in which we are interested, is shown in Figure 6.56. Suppose that user **A**, who has activated *Arc*, is busy with user **B**. If **C**, who has activated *Acb*, attempts to call **A**, he/she would receive a busy signal, then hang up. Since *Acb*(***C***) is active, then the state of **A** is monitored, and when it becomes idle, the following two actions become possible from the initial state of C_1: *acbringsfrom (C, A)* with respect to *Acb* and *arcringsfrom (A, C)* with respect to *Arc*. The choice between these two actions is an internal decision of the controller.

6.2.9.2 Detection of Interactions between Acb and Arc

Figure 6.55 shows a composition, $POTS[B_{acb} |[]| B_{arc}]$, of two partial behaviours, B_{acb} and B_{arc}, of *Acb* and *Arc* in the context of POTS. Two testing processes, t_1 and t_2, are derived from this composition. Executing either of t_1 or t_2 against the integration leads to a deadlock. Consider, for example, the execution of t_1. This sequence forces the integration to synchronize on the action *acbringsfrom* (***C***, ***A***). After this action, the system deadlocks, because t_1 offers *arcringsfrom* (***A***, ***C***) and the integration offers *offhookans*(***C***, ***A***).

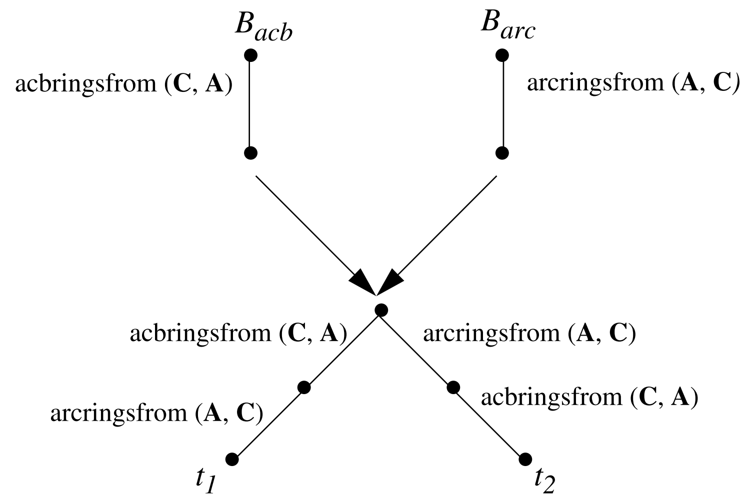


Fig. 6.55 Testing Sequences for *Acb* and *Arc*

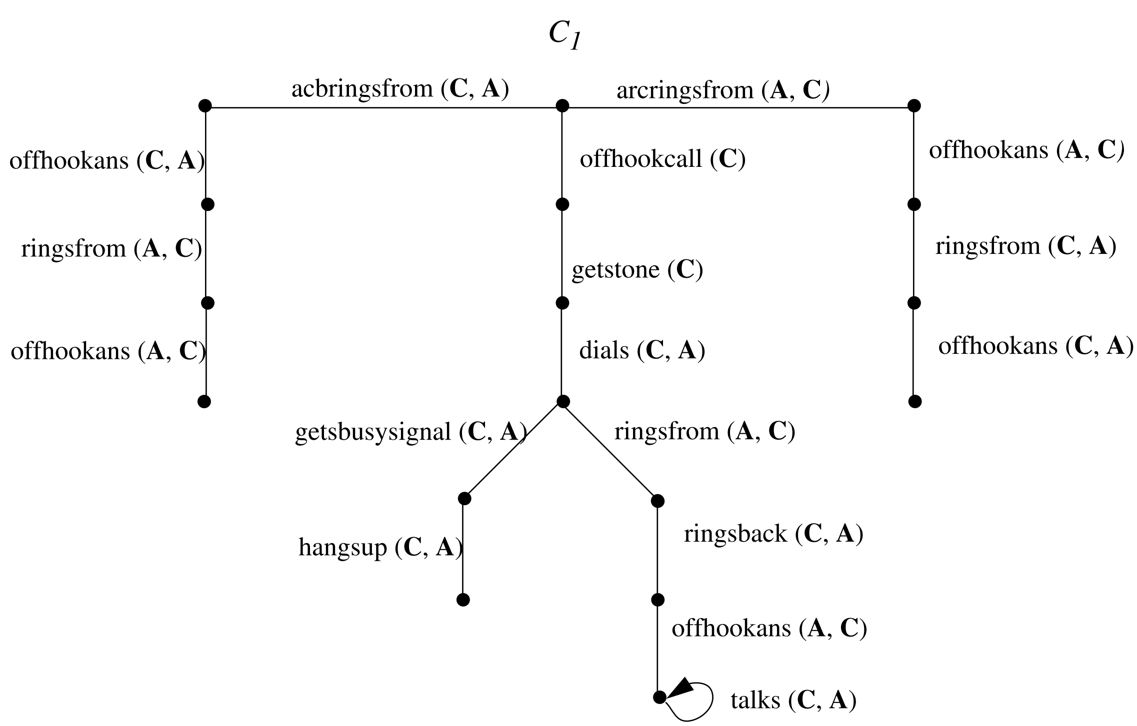


Fig. 6.56 LTS of *Acb* and *Arc* in the context of POTS - MUSE

CHAPTER 7 Conclusion and Future Directions

The feature interaction problem [BDCG89] remains as one the major obstacles for the *rapid development* and introduction of new features into modern telecommunications systems. This thesis describes a methodology, based on a formal approach, for detecting feature interactions at the specification level, which we hope will improve the end product of telecommunications software.

7.1 Summary

The motivation for our work is given in Chapter 1. Chapter 2 presents some formalisms that are used to specify telecommunications software, gives some known classifications of the different types of feature interactions and surveys some of the approaches that are used to solve this problem. Chapter 3 gives a brief overview of the LOTOS language and justifies its use for specifying and designing telecommunications software.

Chapter 4 defines our model for specifying telephone systems and their features. A specification for a single connection for a telephone system consists of three components: a caller side, a called side, and a controller. Processes of the caller side and the called side represent a simple POTS service or a telephone feature. A connection may involve more than one user at a time, and each user may have a basic POTS process executing as well as one or more feature processes. The connections making up the system interleave with each other and fully synchronize with a global constraints process, which exchanges information with all connections in the system.

Each of these processes is expressed in terms of one of the following three types of constraints: (1) *Local constraints* are used to enforce the appropriate sequences of events at each telephone, and are different according to whether the telephone represents a *Caller*, a *Called*, or a *Feature*. Therefore local constraints are represented by different processes, and an instance of each of these is associated with each telephone existing in the system; (2) *End-to-*

End constraints are related to each connection, and enforce the appropriate sequence of actions between telephones in a connection; (3) *Global constraints* are system-wide constraints. They must be satisfied simultaneously over the whole system. Some experimental results that justify these concepts have been obtained by applying the methodology to a simple POTS telephone system [FaLS90], [FaLS91]. The methodology is also practical for structuring more complex telecommunications systems, such as ISDN [ErHM92] and the extension of POTS with features [BoLo93], [FaLo94].

Chapter 5 develops a six-step methodology for detecting feature interactions in telephone systems [FaLo95]. This model is based on the formalization of the concept of *feature interaction*. Intuitively, we say that an interaction exists between *n* features if one of the features cannot exhibit the same behaviour, when integrated into POTS by itself, as it does when integrated in combination with the other features. To formalize the notion of feature interaction, we define the terms *composition* and *integration*. The term *composition*, which we write as $Pots[f_1|[]|f_2]$, expresses the *simultaneous* execution of features. Saying that *n* features execute *simultaneous*ly is equivalent to saying that each feature reaches its terminal states and that, from any given state along the execution path, the features are allowed to synchronize on their common actions with POTS and interleave on their independent actions. The notion of integration, which we express by $Pots[f_1*f_2]$, is defined informally. It expresses the resulting specification of extending a POTS system with *n* features such that each feature is able to execute all the traces which are allowed in the context of POTS, when the other features are disabled. Then, we relate the notion of feature interactions to the concept of conformance in the following way. An interaction exists between *n* features if the *integration* of the features does not **conform** to their *composition*.

Using the testing theory of LOTOS and exploiting the results reported in [BrSS87], [Brin88], [BrAL90], we show that the notion of *conformance testing* has a direct application for detecting feature interactions. Conformance, as defined in the context of Labelled Transition Systems, allows one to reason about two specifications using a single formalism. In our context, the specification representing the *integration (I)* is taken to be an abstract representation of a physical realization and the specification representing the *composition* is taken to be a description of the system's desired behaviour. Therefore, to check whether *I conforms to C* corresponds to the following. Informally, *I* conforms to *C* if testing *I* against the traces of *C* does not lead to deadlocks that would not occur while testing *C* against those same tests. In other words, testing the integration does not produce deadlocks that would not be discovered while testing the composition.

Finally, Chapter 6 presents a case study of concrete telephone examples to show how our methodology can be put into practice. The examples are based on industry-related documents [CGLN94] [Lata91]. In our case studies, we limited ourselves to two types of interactions only: Single User Single Element (SUSE) and Multiple User Single Element (MUSE). In SUSE, we detect the interactions that occur between the features of the same user; in MUSE, we detect the interactions between the features of two different users. For each of the examples, we give an informal description of the features, their formal specifications, their integration, their composition, and finally a testing process that shows how to detect interactions.

Finding solutions to feature interactions can be as simple as suspending the execution of one feature while a second feature is active, or as complex as defining special formalisms, as suggested by [GrVe92], for resolving conflicts at execution time. However, regardless of the resolution approach, we must first develop reliable mechanisms for detection. In this thesis, we have proposed a partial solution, but finding a general solution will not be trivial.

Detecting feature interactions, at the specification level, contributes significantly to speeding up the design phase. Telecommunications systems designers can give precise descriptions and validate their designs as well as devise robust solutions to any potential feature interaction problem before the implementation stage.

7.2 Research Directions

The results obtained in this thesis provide a basis for many future research directions. As new telecommunication services emerge, the need to provide a sound and flexible architecture becomes even greater. We believe that the model we present here for specifying telecommunications features and the formalization of the notion of interactions provides a good starting point for defining such an architecture. Still, there are many ways that other contributions can improve and complement our model.

7.2.1 Merging Specifications

The detection of interactions, in our methodology, is defined with respect to the *integration* of the features under consideration. In our approach, we define a base system to which additional functionalities are added, then we analyse the overall system to check if the functionality of each of the added components is still preserved. Several interesting research directions, under this assumption, are worth investigating. First, we wonder if it would be possible to formalize and *automate* the integration process, i.e., to define a framework where different designers extend the system with different functionalities, then have a system generate the new system which includes the functionalities of all the components. Second, our base system is based on a static architecture, meaning that if the base system is modified, the integrations have to be defined with respect to the new architecture. It would be interesting to study the feasibility of defining a flexible architecture which may be modified while preserving the functionality of existing features.

Other proposals exist for extending the functionality of an existing system. Ichikawa et al. [IcYK90] approach this problem from an incremental point of view, using basic LOTOS specifications. They consider a specification as being tentative, which can be extended by new functionalities using a *merge* operator. Their work is elaborated on in [KhBo93]. However, this notion of extending a system with new functionalities may be used more as a basis for feature interaction *avoidance* than detection, since it is not always possible to extend a system with a new functionality if parts of the existing system constrains the new functionality.

7.2.2 Using Knowledge Goals to Reason about LOTOS Specifications

Another direction that can be explored is to adapt the *knowledge-oriented* model of Halpern and Moses [HaMo90] and incorporate it into LOTOS specifications for telephony systems [FaLo94]. The intuition behind using the knowledge-based approach, is that the designer reasons about LOTOS processes in terms of how relevant information, from the local point of view (i.e., local constraints) of each process, becomes satisfied at certain points during the execution of the system. To analyze whether two features, say *cw* and *cfbl*, interfere with each other, the designer defines a set of knowledge goals and verifies their reachability, when both features are active. If any of the goals is unreachable, the designer concludes that a feature interaction (or design error) exists. Otherwise, no conclusion can be drawn from the analysis. In a way, this is similar to the concept of system testing that we used in our methodology. A test which fails to reveal an error does not indicate that the system under test is error free, it only means that the system is error free with respect to the assumption expressed by the test.

It is interesting to emphasize that, in this knowledge-oriented framework, each process reasons about the outside world only in terms of its local information. Therefore, a process moves from one state to another state based only on its knowledge. Similarly, it gains (or loses) new knowledge as it moves from one state to another. Also, notice that knowledge in this context is an *external* notion, in the sense that processes do not acquire knowledge on their own nor are they able to analyze the knowledge state of other processes. For the purposes of analysis, the specifier is responsible for choosing the appropriate *knowledge goals* used to reason about the system.

7.2.3 Goal-Oriented Exploration of LTSs

The goal-Oriented approach provides another alternative for exploring LOTOS trees. Haj-Hussein et al. [HaLS93] define a new type of inference rules which are capable of generating traces of actions leading to a pre-selected action in the specification. It remains to be seen what flexibility a tool based on the goal oriented approach can provide to the specifier. If a tool is developed such that the specifier can express goals independent of the specification structure and process compositions, then it would provide an invaluable contribution to the domain of detecting feature interactions. For example, if a specifier is able to state goals such as "*find a trace in the specification such that user* ***A*** *is talking to user* ***B*** *and user* ***C*** *is talking to user* ***A***" and *"do all traces lead to a hang up action, from the current behaviour, for user* ***D****?",* then exploring a specification in this manner, i.e., without being forced to mention LOTOS processes or gates explicitly, would be a valuable contribution to our work.

A related direction would be to investigate if a relation can be established between the generation of testing processes from our *composition*, and the definition of goals. Doing so would allow us to reason about the reachability of goals in terms of the ***passes*** relation.The deadlock of a testing process, in the context of our methodology, would be equivalent to a non-reachable goal; analogously, a testing process *passes* the test if the goal is reachable. A non-reachable goal would be interpreted as an interaction, as defined by our formalization.

7.3 Closing Comments

The rigid architecture of existing telephone networks makes the introduction of new features difficult for two reasons: first, each new feature requires the modification of each exchange in the network. Second, the lack of standardization of exchanges which belong to different manufacturers and telephone company operators makes it even more difficult because the same feature software has to be designed and adapted to every type of exchange in the network. Few years ago, the typical time requirements for introducing a new feature was reported to be in the range of three years [Mart88].

To be able to introduce new features more rapidly and to satisfy the ever increasing demands of subscribers, many companies have launched their own projects for building the network of the future. In this context, Bellcore has introduced the Advanced Intelligent Network (AIN)[Bell90]. *AIN is based on a service-independent architecture which aims at supporting the introduction of new features in a flexible and timely manner.* This new architecture may make the introduction of new features easier, but it is unlikely that it will completely solve the *feature interaction problem.*

AIN release 1, Special Report SR-NPL-0011623 (1990), provides a mechanism for managing feature interactions based on a table of rules. The rules are simply a list of prioritized features which are involved in a given pair of (state, event), where the highest priority feature is invoked first. The report does not address the issue of feature interaction detection. It assumes that the possible feature interactions are known in advance, and provides a mechanism for managing them. Therefore, we believe that the methodology we describe in Chapter 5 can be used prior to building such tables.

Bibliography

[Ager79] T. Agerwala, Putting Petri-nets to Work, IEEE Computer, 12, 12, Dec. 1979, 85-94.

[BALL90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans, A Formal Approach to Conformance Testing. In Jan de Meer, Lothar Mackert, and Wolfgang Effelsberg (eds.), Second International Workshop on Protocol Test Systems, 349-363, North-Holland, 1990.

[BDCG89] T.F. Bowen, F.S. Dworak, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin, The Feature Interaction Problem in Telecommunications Systems, 7th International Conference on Software Engineering for Telecommunication Switching Systems, July 1989, 59-62.

[BeHo89] F. Belina and D. Hogrefe, The CCITT Specification and Description Language SDL, Computer Networks and ISDN Systems, Vol. 16, 1989, 311-341.

[BeMD87] D. Belnes, B. Moller-Pedersen and H.P. Dahle, Rationale and Tutorial on OSDL: an Object Oriented Extension of SDL, SDL '87: State of the Art and Future Trends, Proceedings of the Third SDL Forum, in R. Saracco and Tilanus, eds., Leidschendam, The Netherlands, April 1987, 413-426.

[Bell90] Special Report SR-NLP-001623, Advanced Intelligent Network, Release 1, Network and Operations Plan, Issue1, June 1990.

[BiHa85] B. Biebow and J. Hagelstein, Algebraic Specification of Synchronization and Errors: A Telephonic Example. Lectures Notes in Computer Science, Vol. 186, 1985, 294-308.

[BoBr87] B. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59. Also reprinted in [VVD89] 23-73.

[BoLT90] T. Bolognesi, F. Lucidi, S. Trigila, From Timed Petri Nets to Timed LOTOS, in: L. Logrippo, R. Probert, and H. Ural Eds., PSTV X, North-Holland, 1990.

[BoLo93] R. Boumezbeur, L. Logrippo, Specifying Telephone Systems in LOTOS, IEEE Communications Magazine, Aug. 1993, 38-45.

[BoSu80] G. V. Bochmann and C. A. Sunshine, Formal Methods in Communication Protocol Design, IEEE Trans. on Comm., vol. COM-28, no. 4, April 1980, 624-631.

[Boch78] G. V. Bochmann, Finite State Description of Communication Protocols, Computer Networks, vol. 2, 1978, 361-372.

[Boch80] G. V. Bochmann, A General Transition Model for Protocols and Communication Services. IEEE Trans. Comm., 28 (1980), 643-650.

[Boum91] R. Boumezbeur, Design, Specification, and Validation of Telephony Systems in LOTOS, Master's Thesis, 1991.

[BrAt94] K. Braithwaite and J. Atlee, Towards Automated Detection of Feature Interactions, Second International Workshop on Feature Interactions in Telecommunications

Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 36-59.

[BrSS87] E. Brinksma, G. Scollo, and C. Steenbergen, LOTOS Specifications, their implementations, and their Tests. In: B. Sarikaya and G.V. Bochmann (Eds.), Protocol Specification, Testing, and Verification, VIII, North-Holland, 1987, 349-360.

[Bri88a] E. Brinksma, A Theory for the Derivation of Tests. In: Aggarwal, S., and Sabnani, K., (Eds.) Protocol Specification, Testing, and Verification, VIII, North-Holland, 1988, 63-74.

[Bri88b] E. Brinksma, On the Design of Extended LOTOS, A Specification Language for Distributed Systems. Doctoral Dissertation, Universiteit Twente (NL), 1988.

[BrTV91] E. Brinksma, J. Tretmans, L. Verhaard, A Framework for Test Selection, In B. Jonsson, P. Parrow, and B. Pehrson, eds., Protocol Specification, Testing, and Verification IX. North-Holland, 1991.

[CGLN94] E. J. Cameron, N. Griffeth, Y. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen, A Feature Interaction Benchmark for IN and Beyond, IEEE Communications, vol. 31, No. 3, March 1993, 64-69. Aslo reprinted in [Fits94].

[CHCK89] C.J. Chung, J.P.Hong, W.Choi,H.K.Kim, and Y.K.Lee, Using SDL in Switching System Development, SDL ‘89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 377-386.

[CaLi91] E. J. Cameron and Y.J. Lin, A Real-Time Transition Model for Analyzing Behavioral Compatibility of Telecommunications Services. In Proceedings of the ACM SIGSOFT 1991 Conference on Software for Critical Systems, December 1991, New Orleans, Louisiana, 101-111.

[Cass93] Christos Cassandras, Discrete Event Systems: Modeling and Performance Analysis, Richard D. Irvin, Inc., and Aksen Associates, Inc., 1993.

[CaVe93] J. Cameron and H. Velthuijsen, Feature Interactions in Telecommunications Systems, IEEE Communications Magazine, Aug. 1993, 18-23.

[CaVu92] R. C. Cam and S. T. Vuong, A Formal Specification, in LOTOS, of a Simplified Cellular Mobile Communication System, Forte 92, 628-642.

[Cain92] M. Cain, Managing Run-Time Interactions Between Call-Processing Features, IEEE Communications Magazine, February 1992, 44-50.

[ChJa89] K.E. Cheng and L.N. Jackson, A Definition and Description Technique for Translating SDL Specifications to Implementation, SDL ‘89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 293-302.

[Chen94] K. E. Cheng, Towards a Formal Model for Incremental Service Specification and Interaction Management Support, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H.

Velthuijsen, ISO Press 1994, 152-166.

[Chen94] K. E. Cheng, Towards a Formal Model for Incremental Service Specification and Interaction Management Support, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 152-177.

[Comp93] IEEE Computer, Special Issue on Feature Interactions in Telecommunications Systems, Aug. 1993.

[DaNa93] O. Dahl and E. Najm, Specification and Detection of IN Service Interference Using LOTOS, Proceedings Forte '93, eds. R. L. Tenney, P. D. Amer, M. U. Uyar, Boston 1993, 53-71.

[Davi88] A. M. Davis, A Comparison of Techniques for the specification of external system behaviour, Communications of the ACM, Vol. 31, No. 9, Sept. 1988, 1098-1115.

[Dwor91] F. S. Dworak, Approaches to Detecting and Resolving Feature Interactions, GLOBECOM 1991, 1371-1377.

[EKDB92] M. Erradi, F. Khendek, R. Dsouli, and G. V. Bochmann, Dynamic Extension of Object-Oriented Distributed System Specifications, First International Workshop on Feature Interactions in Telecommunications Software Systems, Florida, 1992, 116-132.

[EhMa85] H. Ehrig and B. Mahr, Fundamentals of Algebraic Specification 1, Springer-Verlag, Berlin, 1985.

[ErFa94] M. Erradi, M. Faci, Evolution des Specifications de Systemes Distribues: Cas d'un systeme telephonique Simplifie. Third Maghrebian Conference on Software Engineering and Artificial Intelligence, April 1994.

[ErHM92] P. Ernberg, T. Hovander, and F. Monfort, Specification and Implementation of an ISDN Telephone System Using LOTOS, Forte92, 179-194.

[FWFI92] The First International Workshop on Feature Interactions in Telecommunications Software Systems, Florida, 1992.

[FaLS90] M. Faci, L. Logrippo and B. Stepien, Formal Specifications of Telephone Systems in LOTOS. In E. Brinksma, G. Scolo, and C. Vissers, eds., Protocol Specification, Testing, and Verification IX. North-Holland, 1990.

[FaLS91] M. Faci, L. Logrippo and B. Stepien, Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach, Computer Networks and ISDN Systems, 21, North Holland, 1991, 52-67.

[FaLo93] M. Faci and L. Logrippo, Specifying Hardware in LOTOS, Proceedings of the IFIP WG 10.2 11th International Conference on Computer Hardware Description Languages and their Applications, Ottawa, Canada, April 1993, 305-312.

[FaLo94] M. Faci and L. Logrippo, Specifying Features and Analysing their Interactions in a LOTOS Environment, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO

Press 1994, 136-151.

[FaLo95] M. Faci and L. Logrippo, An Algebraic Approach for Detecting Feature Interactions, technical report, Computer Science, University of Ottawa, 1995.

[FiIm86] M. J. Fischer and N. Immerman, Foundations of Knowledge for Distributed Systems. In Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference, J. Y. Halpern, ed. Morgan Kaufmann, Calif., 1986, 171-186.

[Fits94] Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994.

[FuWW91] M. Fukazawa, M. Wakamoto, and M. Wan Kim, Intelligent Network Call Model for Broadband ISDN, ICC 1991, 964-968.

[Gell87] P. Gelli, Evaluation and Comparison of Three Specification Languages: SDL, LOTOS and Estelle, SDL '87: State of the Art and Future Trends, Proceedings of the Third SDL Forum, in R. Saracco and Tilanus, eds., Leidschendam, The Netherlands, April 1987, 211-232.

[GrVe92] N. D. Griffeth and H. Velthuijsen, Conflict Resolution in Telecommunications Systems: A Formalism for Proposal Generation, 1992.

[GrVe94] N. D. Griffeth and H. Velthuijsen, The Negotiating Agents Approach to Runtime Feature Interaction Resolution, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 217-235.

[GuHL88] R. Guillemot, M. Haj-Hussein, and L. Logrippo, Executing Large LOTOS Specifications. In Aggarwal, S., and Sabnani, K., (eds.) Protocol Specification, Testing and Verification, VIII, North-Holland, 1988, 399-410.

[GuLo89] R. Guillemot and L. Logrippo, Derivation of Useful Execution Trees from LOTOS by Using an Interpreter. In: Turner, K. J. (ed.), Formal Description Techniques, North- Holland, 1989, 311-325.

[HVNM89] S. O. Hallsteinsen, A. Venstad, A. Nyeng, and H. Martinsen, Transformational Program Development - An Approach for Translating SDL to CHILL, SDL '89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 283-292.

[HaFa89] J. Y. Halpern and R. Fagin, Modelling Knowledge and action in distributed systems, Distributed Computing, 3, 1989, 159-177.

[HaLS93] M. Haj-Hussein, L. Logrippo, and J. Sincennes, Goal Oriented Execution for LOTOS. In: M. Diaz and R. Groz (eds.) Formal Description Techniques, V. North-Holland, 1993. 311-327.

[HaLo91] M. Haj-Hussein and L. Logrippo, Specifying Distributed Algorithms in LOTOS, Technical Report TR-91-04, Computer Science, Univ. of Ottawa, May 1991.

[HaMo90] J. Y. Halpern and Y. Moses, Knowledge and Comon Knowledge in a Distributed Environment, JACM, Vol. 37, No. 3, July 1990, 549-587.

[Hare87] D. Harel, Statecharts: A Visual Formalism for Complex Systems, Sci. Comp. Prog. 8, 1987.

[HiHT90] Y. Hirakawa, Y. Harada, and T. Takenaka, Behaviour Descriptions for a System which Consists of an infinite Number of Processes, Bilkent International Conference on New Trends in Communication, Control, and Signal Processing, ed., Erdal Arikan, Ankara, Turkey, July 1990.

[HoSi88] S. Homayoon and H. Singh, Methods of Addressing the Interactions of Intelligent Network Services With Embedded Switch Services, IEEE Communications Magazine, Dec. 1988, 42-47.

[Hoar85] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[ISO87] Information Processing Systems, Open Systems Interconnection. Estelle, A Formal Description Technique Based on an Extended State Transition Model, International Organization for Standardization, DIS 9074, July 1987.

[ISO88a] International Organization for Standardization, Information Processing Systems, Open Systems Interconnection. IS 8807: LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (1988).

[ISO88b] ISO/IEC JTC1/SC6 N4870, Formal Description of ISO 8072 (Transport Service) in LOTOS (working draft), 1988.

[ISO89] Information Processing Systems, Open Systems Interconnection. Guidelines for the Application of Estelle, LOTOS and SDL, ISO/IEC JTC 1/SC21 N 3252, June 1989.

[ISO90] ISO/IEC - Information Processing Systems Open Systems Interconnection, Architectural Semantics, Specification Techniques and Formalisms, ISO/IEC JTC1/SC21/WG 7, N314, December 1990.

[ISO91] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646, 1991.

[IcYK90] H. Ichikawa, K. Yamanaka and J. Kato, Incremental Specification in LOTOS, Tenth International IFIP Symposium on Protocol Specification, Testing and Verification, in L. Logrippo, R.L. Probert, and H. Ural eds., Ottawa, Canada, June 1990, 185-200.

[Inoue92] Y. Inoue, K. Takami, and T. Ohta, Method for Supporting Detection and Elimination of Feature Interaction in a Telecommunication System, First International Workshop on Feature Interactions in Telecommunications Software Systems, Florida, 1992, 61-81.

[Jens87] K. Jensen, Coloured Petri Nets, Lecture Notes in Computer Science, Vol. 254, 1987, 248-299

[Kara89] G. M. Karam, Mlog: A Language for Prototyping Concurrent Systems, TR SCE-88-3, Dept. of Systems and Computer Eng., Carleton University, May 1989.

[Kawa71] H. Kawashima et al. Functional specification of call processing by state transition diagram. IEEE Trans. on Comm. 19, 5, Oct. 1971, 581-587.

[KhBo93] F. Khendek and G. V. Bochmann, Incremental Construction Approach for Distributed System Specifications, Proceedings Forte ‘93, eds. R. L. Tenney, P. D. Amer, M. U. Uyar, Boston 1993, 89-104.

[LOBF88] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri An Interpreter for LOTOS, a Specification Language for Distributed Systems. Software-Practice and Experience, 18 (1988) 365-385.

[Lata89] LATA Switching Systems Generic Requirements (LSSGR), Bellcore, TR-TSY-000064, FSD 00-00-0100, July 1989.

[Ledu92] G. Leduc, An Upward Compatible Timed Extension to LOTOS, in: K. Parker, G. Rose3 eds., FORTE‘91, North-Holland, 1992.

[Lee92] A. Lee, Formal Specification and Analysis of Intelligent Network Services and their Interaction, Ph. D. Thesis, Dept. of Computer Science, University of Queensland, 1992.

[LiLi94] F. J. Lin and Y-J. Lin, A building Block Approach to Detecting and Resolving Feature Interactions, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 86-119.

[Lin90] Y. J. Lin, Analyzing Service Specifications Based upon the Logic Programming Paradigm, In Proceedings of the IEEE GLOBECOM 1990, December 1990, Sandiego, California, 651-655.

[Lin93] Y. J. Lin, Bellcore, New Jersey, U.S.A., Private Communication, 1993.

[LoFH92] L. Logrippo, M. Faci and M. Haj-Hussein, An Introduction to LOTOS: Learning by Examples, Computer Networks & ISDN Systems, Vol. 23, No. 5, 1992, 325-342.

[MaMi89] J. A. Manas and T. de Miguel-More, From LOTOS to C. In: K.J.Turner (ed.) Formal Description Techniques, North-Holland, 1989, 79-84.

[Magz93] IEEE Communications Magazine, Special Issue on Feature Interactions in Telecommunications Systems, Aug. 1993.

[Mart88] R. L. Martin, Future Telecommunications Services, IEEE Global Telecommunications Conference, 1988, 721-725.

[MiTJ93] J. Mierop, S. Tax, R. Janmaat, Service Interaction in an Object Oriented Environment, IEEE Communications Magazine, Aug. 1993.

[Miln80] R. Milner, A Calculus of Communicating Systems. Lecture Notes in Computer Science No.92 (Springer-Verlag) 1980.

[Miln89] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[Moll87] B. Moller-Pederson, D. Belsnes and H. P. Dahle, Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL, Computer Networks and ISDN Systems, Vol. 13, No.2, 1987, 97-117.

[More89] R. Moretti, SDL and Object Oriented Design: A Way for Producing Quality Software, SDL ‘89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 387-394.

[NgJP89] H. T. Nguyen, L. N. Jackson and K. R. Parker, Reachability Graph generator for SDL, SDL '89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 219-230.

[NiLM89] G. Nilsson, I. Ljungdahl, and P. Madsen, SDL Toolbox to Support Different SDL Environments, SDL '89: The Language at Work, Proceedings of the Fourth SDL Forum, in O. Faergemand and M. M. Marques, eds., Lisbon, Portugal, October 1989, 87-94.

[NuPr93] K. Nursimulu and R. L. Probert, Cause-Effect Validation of Telecommunications Service Requirements, Technical Report TR-93-15, University of Ottawa, Dept. of Computer Science, October 1993.

[OSDL92] CCITT, Recommendation Z.100, Specification and Description Language SDL, Study Group X, 1992.

[OhHa94] T. Ohta and Y. Harada, Classification, Detection and Resolution of Service Interactions in Telecommunication Services, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 60-72.

[PaTa92] P. Panangaden and K. Taylor, Concurrent Common Knowledge: Defining Agreement for Asynchronous Systems, Distributed Computing, 6, 1992, 73-93.

[Pete77] J. Peterson, Petri-nets, ACM Computing Surveys, 9, 3, Sept. 1977, 223-252.

[QuAF90] J. Quemada, A. Azcorra, D. Frutos, A Timed Calculus for LOTOS, in: S.T. Vuong ed., FORTE'89, Vancouver, Canada, Dec. 89, North-Holland, 1990.

[QuAz89] J. Quemada and A. Azcorra, A Constraint-Oriented Specification of Al's Node. In [VVD], 1989, 83-88.

[QuFe87] J. Quemada and A. Fernandez, Introduction of Quantitative relative Time in LOTOS. In: Rudin, H., and West, C.H. (eds.) Protocol Specification, Testing, and Verification, VII. North-Holland, 1987, 105-121.

[QuPF89] J. Quemada, S. Pavon, and A. Fernandez, Transforming LOTOS Specifications with LOLA - The Parameterized Expansion. In: Turner, K. J. (ed.), Formal Description Techniques, North-Holland, 1989, 45-54.

[RaRS91] A. S. Ramani, J. R. Rosenberg, and P. B. Shanghavi, Placement of Feature Management functionality in IN, ICC 1991, 632-636.

[RoKa86] S. J. Rosenschein and L. P. Kaelbling, The Synthesis of Digital Machines with Provable Epistemic Properties. In Theoretical Aspects of Reasoning about Knowledge: Proceedings of the 1986 Conference, J. Y. Halpern, ed. Morgan Kaufmann, Calif., 1986, 171-186.

[Ryan91] M. G. Ryan, Software Application and Management issues for AIN, ICC 1991, 632-636.

[SDL88] CCITT Recommendation Z.100: Specification and Description Language, 1988.

[Scho90] J. Schot, The Role of Architectural Semantics in the Formal Approach of Distributed Systems Design, Ph. D Thesis, Twente University, 1990.

[Sedo87] SEDOS: Public Report - Task C1, Deliverable SEDOS/119, ESPRIT Project 410, Commission of the European Communities, Brussels, October 1987.

[Simo89] L. Simon, An Object-Oriented Delta Approach to Telecommunication Service Specification, Master's Thesis, Dept. of Systems and Computer Engineering, Ottawa- Carleton Institute for Electrical Engineering, 1989.

[TsMa94] S. Tsang and E. H. Magill, Detecting Feature Interactions in the Intelligent Network, Second International Workshop on Feature Interactions in Telecommunications Software Systems, eds. L. G. Bouma and H. Velthuijsen, ISO Press 1994, 236-248.

[Tur88a] K. Turner, Constraint-Oriented Style in LOTOS. In: Proc. of the British Computer Society Workshop on Formal Methods in Standards, Didcot, April 1988.

[Tur88b] K. Turner, The Formal Specification Language LOTOS: A Course for Users, Course notes, University of Stirling, June 1988

[Turn87] K. Turner, An Architectural Semantics for LOTOS. In: Rudin, H., and West, C. H. (eds.) Protocol Specification, Testing, and Verification, VII. North-Holland, 1987, 15-28.

[Tvrd89] I. Tvrdy, Formal Modelling of Telematics Services Using LOTOS, Microprocessing and Microprogramming, Vol. 25, 1989, No. 313-317, 1-5.

[VPGW91] P. H. Vapheas, B.A. Polonsky, A.M. Gopin, and R.J. Wojcik, Advanced Intelligent Network: Evolution, ICC 1991, 941-947.

[VSVB91] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, Specification Styles in Distributed Systems Design and Verification, Theoretical Computer Science 89, 1991, 179-206.

[VaTZ90] W. van Hulzen, H. Tilanus and H. Zuidweg, LOTOS Extended with Clocks, to appear in: Vuong, S., (ed.) Formal Description Techniques, North-Holland, 1990.

[VaVD89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz (eds.). The Formal Description Technique LOTOS. North-Holland, 1989.

[Vane88] P. van Eijk, Software Tools for the Specification Language LOTOS. Doctoral Thesis, Universiteit Twente (1988).

[ViLo86] C. A. Vissers and L. Logrippo, The Importance of the Service Concept in the Design of Data Communications Protocols. In Diaz, M. (ed.) Protocol Specification, Testing, and Verification, V. North-Holland, 1986.

[ViSV88] C. A. Vissers, G. Scollo, and M. van Sinderen, Architecture and Specification Style in Formal Descriptions of Distributed Systems. In: Aggarwal, S., and Sabnani, K., (eds.) Protocol Specification, Testing and Verification, VIII, North-Holland, 1988, 189-204.

[Viss89] C. A. Vissers, LOTOS Background, in van Eijk, Vissers, C. A. and Diaz M. (eds) The Formal Description Technique LOTOS, North-Holland, 1989, 15-22.

[WhCh81] V. Whitis and W. Chiang, A State Machine Development Method for Call Processing Software. In Proceedings of the IEEE Electro 81, IEEE Press, Wash. D.C., April 1981.

[YoBa77] M. Yoeli and Z. Barzilai, Behavioural Descriptions of Communications Switching Systems using Extended Petri-nets, Digital Processes 3, 4(1977), 307-320.

[ZaJa91] P. Zave and M. Jackson, Techniques for Partial Specification and Specification of Switching Systems, Lecture Notes in Computer Science, 551, eds. S. Prehn and W.J. Toetenel (VDM'91: Formal Software Development Methods), 1991, 511-525.

[ZaJa93] P. Zave and M. Jackson, Conjunction as Compositions, ACM Transactions on Software Engineering and Methodology, 2(4):379-411, October 1993.

[ZaJa94] P. Zave an M. Jackson, Where do Operations Come From? A Multiparadigm Specification Technique, available from the authors, March 94.

[Zave85] P. Zave, The Distributed Alternative to Finite-State-Machine Specifications. ACM Trans. On Prog. Lang. and Systems, Vol. 7, No. 1, Jan. 1985, 10-36.

[Zave89] P. Zave, A Compositional Approach to Multiparadigm Programming, IEEE Software 6, 5, September 1989, 15-25.

[Zave93] P. Zave, Feature Interactions and Formal Specifications in Telecommunications, IEEE Computer, Aug. 1993, 20-30.