Formal Methods for Telecommunication System Requirements: A Survey of Standardized Languages

Mark A. Ardis
Bell Laboratories
Naperville, IL 60566-7013
maa@research.bell-labs.com

Abstract

Modern telecommunications systems are so complicated that informal languages are no longer sufficient for expressing their requirements. A brief introduction to the nature of requirements for telecommunication systems is given in this paper. The three standardized formal languages for telecommunications, Estelle, LOTOS, and SDL, are described and compared. Each language is evaluated, and a comparison of all three is offered. Throughout the paper a common example is used.

(Appeared in *Annals of Software Engineering* 3 (1997) 157-187.)

1. INTRODUCTION

Telecommunication systems are becoming more and more complex. It is no longer possible to give a reasonably complete and consistent set of requirements in natural language. We need languages that allow us to check our specifications for errors with automated tools.

The telecommunications industry has attempted to solve this problem by proposing their own languages and methods. Specifically, they have created and standardized 3 formal languages: Estelle, LOTOS, and SDL. A fourth notation, Message Sequence Charts, has been derived from SDL. These technologies borrow from the success of other formalisms and accrue most of their benefits. But, there are still problems that these methods do not solve.

This survey describes the three standard languages and compares their features. A simple example drawn from a telecommunications standard is described in each language. In addition to commentary on the particular strengths and weaknesses of each language, this paper offers a comparison across a set of criteria that emphasizes industrial value.

Section 2 describes the types of problems that must be solved in order to effectively describe requirements in this area. Section 3 surveys the three standard languages Estelle, LOTOS, and SDL. Section 4 compares the languages. Finally, Section 5 offers an evaluation of the accomplishments of these standard languages in general, and suggest remaining problems to be solved.

2. TELECOMMUNICATION SYSTEMS REQUIREMENTS

Telecommunication systems are a rich area for requirements specification problems. There are at least two types of behavior that need to be described in these systems:

- Users and Systems: the interaction between users and devices
- Systems and Systems: the interaction of telecommunication systems with one another

2.1 Users and Systems

Many years ago a telephone was a simple device to operate. Even if one still uses a simple device one can subscribe to an assortment of features, such as call forwarding, call waiting, automatic callback, automatic recall, etc. The explosive growth of all these features has created a problem for their providers, namely how to describe them.

A standard method for specifying telephone features is by examples, or scenarios. Unfortunately, this is like describing a system by enumerating test cases.

2.1.1 Example

Automatic Callback (AC) is a feature provided by most local telephone companies today. Briefly, this feature enables you to (automatically) keep checking the busy status of a phone until that phone becomes available. When that happens, a call is attempted. From the viewpoint of the user of the feature, it seems easy to describe:

1. You call someone, but their phone is busy.
2. You hang up, then pick up your phone and dial an access code for AC.

3. You hear a confirmation tone and then you hang up.
4. Eventually the other party becomes available.
5. Your phone rings with a special signal.
6. You pick up your phone.
7. A call to the other phone is automatically attempted.

This simple scenario is only one of many that might occur. In order to completely describe the feature one must consider several other possibilities, such as:

- What happens when more than one party attempts to use this feature for the same destination?
- What if the user of the feature does not pick up when they are signaled that the other party is free?
- What if the other party becomes busy while the originating party is being alerted?
- What if the originating party wants to cancel the AC attempt before it is finished?

2.1.2 Formalizing the Example

Scenarios have the advantage that they show real examples of usage. What is needed, however, are more complete descriptions of feature behavior. If scenarios could be derived from these descriptions, then users would be able to test their understanding by postulating test cases. In addition to testing such descriptions, it is important to perform more complete analysis to ensure that safety and liveness properties are satisfied.

For the Automatic Callback example, what is needed is a complete model of the behavior of each of the processes and their interactions. Figures 1, 2 and 3 show part of such a model using a finite state machine paradigm. In each figure, states are represented by ovals. Each transition may be labeled with an input trigger, and an output action, separated by a slash. Either the input or the output may be absent, but not both.

Figure 1 shows the automaton for the calling party. The process starts in the **Idle** state. After the customer decides to initiate the automatic callback (**ac**) feature, the process enters the **Waiting** state. Once there, the only allowed stimulus is the ringing of the phone, shown by the **ringa** event.

If the customer answers the phone (**answera**) the process moves to the **Calling** state. If the customer does not answer (**noanswera**), the process returns to the **Idle** state. From the **Calling** state the customer may hang-up (**hangupa**) before the call is completed, receive a busy signal (**busytonea**), or be connected to the other party (**connect**).

If the customer hangs up the process returns to the **Idle** state. If **busytonea** is received the process returns to the **Waiting** state to try again. If a connection is made the process enters the **Talking** state. The only allowed transition from **Talking** is to **Idle**.

Figure 2 shows the automaton for the switching system agent. Initially the system is in the **Idle** state. When it receives an automatic callback request (**ac**) it checks to see if the other party is available (**checkb**). If the party is available (**available**) the agent rings the calling party (**ringa**) and enters the **RingingA** state. If the called party is not available (**busyb**), the agent enters the **Camping** state. From there it checks on the called party (**checkb**) and re-enters the **Checking** state.

Once the process has entered the **RingingA** state, it can either detect an answer (**answera**) or not (**noanswera**). In the first case it rings the called party (**ringb**) and enters the **RingingB** state. In the second case it simply returns to the **Idle** state.

From the **RingingB** state there are three possibilities: the called party answers (**answerb**), the called party returns busy (**busyb**), or the calling party hangs up prematurely (**hangupa**). If the called party answers the agent signals a connection (**connect**) and returns to the **Idle** state. If the called party is busy the agent sends a busy signal (**busytonea**) to the calling party and returns to the **Camping** state to try again. If the calling party hangs up before the connection is made the agent stops ringing the called party (**stopringb**) and returns to the **Idle** state.

Figure 3 shows the automaton for the called party. This automaton has only 3 states: **Onhook**, **Offhook**, and **Ringing**. From the **Onhook** state the process may spontaneously emit the **offhook** output and enter the **Offhook** state. Similarly, it may spontaneously move from **Offhook** to **Onhook**. If the process receives the **checkb** signal when in the **Onhook** state it emits **available** and stays **Onhook**. Likewise, if it receives **checkb** or **ringb** from the **Offhook** state it remains in that state and returns **busyb**. But, if it receives **ringb** when in the **Onhook** state, it moves to the **Ringing** state. From there it may receive **stopringb**, indicating that the other party hung up early, in which case it returns to **Onhook**. Otherwise, the phone is answered (**answerb**), moving to **Offhook**.

2.2 Systems and Systems

Modern telecommunication systems derive their value from their connection to a network of other systems. For example, one would not be happy with a telephone that could only communicate with telephones that were connected to the same local office switch.

The telecommunications industry has developed a large collection of standards for system interaction. Most of these standards describe protocols.

2.2.1 Example

Automatic Protection Switching (APS) [Bellcore 1991] is a protocol designed to increase reliability of a communication link. The idea is to provide more than one line for each communication channel. If a line degrades or fails, a backup line, called the **protection line**, is used instead. The simplest version of this protocol is **1+1 unidirectional non-revertive** APS. In this strategy, a protection line is allotted for each working line (1+1), the decision to switch lines is only made by the receiving side (unidirectional), and a switch to the protection line remains in effect even after the working line clears to an equivalent condition (non-revertive).

A standard redundancy method is used to check the accuracy of transmission of messages. We can assume that the number of erroneous bits received on the working line is continuously recorded, and that correction of messages is not an issue. (Some other protocol will take care of repair or retransmission of faulty messages.)

A line signal is considered **degraded** when it has a bit error rate within a dangerous range, typically between 10^{-5} and 10^{-9}. A line signal is considered to have **failed** when the bit error rate exceeds the degraded range, or whenever other hard failures have occurred, such as a complete loss of signal. (Since the signal is continuously monitored, a complete loss of signal is first detected as a degraded condition, and then a failure.) Either a degraded or failed line may **clear** spontaneously. The expected response to a degraded or failed signal on the working line is to (automatically) switch to the protection line, if that line is in better condition.

Technicians are provided with a set of commands to change the configuration of the channel:

- remove line: The line is taken out of service.
- restore line: The line is placed in service.
- forced switch: The specified line is selected for communication, as long as it is in service.
- conditional switch: The specified line is selected for communication, as long as it is available and of at least the quality of the selected line.

A protocol is needed to maintain the highest quality communication available while responding to operator requests and signal degradation and failure.

2.2.2 Formalizing the Example

One way to present a protocol is to list all the possible states and transitions of the system in a table. Figure 4 shows such a table.

The states of the system are represented by a 3-letter code. The first letter indicates which line is selected (W for the working line, P for the protection line). The second letter shows the status of the working line and the third letter shows the status of the protection line. The status of a line may be normal (N), degraded (D), failed (F), or Out of Service (O).

Input events are shown by 3-letter codes, also. The first two letters show the event type and the third letter indicates which line (W or P) is affected. Event types are remove (RM), restore (RS), forced switch (FS), conditional switch (CS), degrade (DG), fail (FL), or clear(CL).

2.3 Problems

When describing systems that interact with users it is reasonable to draw finite state diagrams for small systems, but this technique quickly becomes unwieldy as the systems grow in size and complexity. Modern telecommunication features have far too many behaviors to describe this way. What is needed are languages and methods that allow for decomposition of the problem into manageable pieces.

The same is true for specification of systems interacting with other systems. Although it is possible to specify the simplest version of APS with a state transition table, more complicated versions cannot be reasonably handled this way. (Even this simple example would be unwieldy if not for the symmetry of behavior of the two lines. This allows for a compact notation to show status and events.) The bidirectional version of the protocol requires several hundred states. The APS standard does not even include the table. Instead, a few of the interesting scenarios are described, and a set of rules are provided to interpret the rest of the cases. Unfortunately, the rules are not adequate to unambiguously decide all of the cases.

Formal methods, then, may offer hope to those who simply want to understand what a given system is supposed to do. But, there are other reasons for wanting precision in these requirements specifications. Interoperability is an important concern. Given that different vendors may be supplying the equipment at different ends of a protocol, how can one be sure that they have both interpreted the standard in a compatible way?

3. FORMAL METHODS FOR TELECOMMUNICATION

3.1 Help from Formal Methods

Formal methods offer some help for the problems mentioned. Specifically, formal methods claim to provide:

- languages to describe behavior precisely and unambiguously,
- analysis methods and tools to reveal errors.

Some methods also provide a means to execute, or simulate the behavior of the systems described.

Historically, the telecommunications industry has adopted its own standards and languages, separate from the rest of software engineering. The CCITT (International Consultative Committee on Telegraphy and Telephony), now the ITU-T (International Telecommunication Union), established its own languages for specification and implementation of systems. Requirements for many international protocols are specified in these languages.

Later, the ISO (International Organization for Standardization) sponsored work to invent new languages to specify the OSI (Open Systems Interconnection) protocols. The ITU-T and ISO standardized languages are known as Formal Description Techniques. There are three of these languages that were standardized in the 1980's: Estelle[ISO 1989a], LOTOS[ISO 1989b], and SDL[ITU 1993].

3.2 Estelle

Estelle was developed in the 1980's by the ISO FDT group. It is based on finite state automata, and borrows heavily from the syntax of Pascal. In addition to specifying the behavior of communicating processes, one can describe the interconnections between processes, which may change as the processes run.

3.2.1 Language Overview

Each process in Estelle is a finite transducer [Aho and Ullman 1972]. That is, each transition is triggered by some input, and it may also generate output. Inputs correspond to messages received by the process, and outputs correspond to messages sent to other processes. In fact, the messages may be sent from a process to itself, though that is not a common technique.

The basic unit of process description is a **module**. Modules communicate with one another via **channels**, which link the output of one module to an unbounded input queue of another module. These links are called **interaction points**. The messages that pass between modules are called **interactions**.

3.2.1.1 Channels

Estelle processes communicate by passing messages through channels. Each channel type is specified by enumerating the messages that may pass through it. Messages may have parameters, so the specification only describes the structure of the messages. An instance of a channel is created by specifying its interaction points. This is either done in a module header or module body.

In the example in Figure 5, a channel type **Buy** is described that allows **coin** events to pass from a **Customer** to a **Vendor**, and which allows **trinket** events to pass from the **Vendor** to the **Customer**. One instance on the channel is created by specifying the interaction points **p1** and **p2**.

3.2.1.2 Modules

In Estelle a system consists of a hierarchy of interacting modules. Each parent module controls the creation and destruction of its children modules. A module may interact with another module at the same level in the hierarchy, or with its own children. (There is also a mechanism for passing interactions through from one level to another.)

The specification of a module consists of two parts: the module header and a list of possible module bodies. In the header one enumerates the external interaction points and exported (shared) variables. The header also declares the module to be either a process (which executes concurrently with other processes) or an activity (which can only run in sequence with the other activities at its level in the module hierarchy).

A module body consists of three subparts: declaration, initialization, and transition. The declaration part contains Pascal-style declarations of variables and procedures and declarations of Estelle objects (channels, modules, module variables, states, and internal interaction points). The initialization part creates new instances of submodules and establishes connections between them. The transition part specifies the behavior of the module in response to its interactions.

Creation of a module instance is accomplished with the **init** action. Instances may be destroyed with the **release** action.

The information about interaction points contained in a module header describes the type of interactions allowed. But, connections between interaction points is accomplished by **connect** and **attach** statements. The difference between these two statements is that **connect** establishes a new communication channel, whereas **attach** continues a channel from a parent to a child. An attached channel passes interactions through from parent to child (or vice versa), so that the other module may process them. To break the connection one uses **disconnect** or **detach**, respectively.

3.2.1.3 Transitions

Each transition in Estelle has a **condition** and an **action**. The condition specifies whether the transition is enabled. It may be described in terms of one or more of the following types of clauses:

- from
- when
- provided
- priority
- delay

The **from** clause type specifies a set of states in which the transition is enabled. The **when** clause specifies an interaction (input message) that must be present at the front of the input queue. The **provided** clause specifies a Boolean expression that must be true. The **priority** clause specifies a non-negative integer priority to associate with the transition. The **delay** clause specifies

lower/upper bounds of time units that must/may elapse before the transition is enabled.

Clauses may be omitted, and at most one clause of each type is allowed. The transition is enabled if all of its clauses are satisfied, and it has the lowest valued priority of all transitions currently enabled. Nondeterminism is allowed, and implementations are free to choose any allowed transition in that case.

The transition action has two parts: the name of the next state, and an optional block of Pascal statements to execute after firing the transition. The block of statements allows the creation and destruction of module instances (processes), the connection and disconnection of communication channels, and the sending of interactions (messages).

The local variables used in Estelle reduce the number of states that must be explicitly enumerated in the specification, but they do not reduce the effective state space of the system. For example, a Boolean variable doubles the size of the state space, because the variable may be either true or false in each named state of the process. These variables may also be used to describe internal computations and to simplify control statements associated with transitions.

3.2.2 Example

Figures 6- 9 illustrate Estelle via the AC example. Two channels are described. Channel **CA** is for communication between the calling party (**PartyA**) and the switching system (**Agent**). Channel **CB** is for communication between the switching system and the called party (**PartyB**).

Channel **CA** allows the calling party to send four types of messages to the switching system: activate automatic callback (**ac**), answer a call (**answera**), hang-up a call (**hangupa**), and fail to answer (**noanswera**). This last message is a little unusual, since it really represents the absence of a message. A more complete example would use a timer to represent this phenomenon.

Channel **CA** allows the switching system to send three types of messages to the calling party: busy signal (**busytonea**), connection established (**connect**), and ringing (**ringa**).

Channel **CB** allows the called party to send three types of messages: answer the phone (**answerb**), reply that the party is available (**available**), and reply that the party is not available (**busyb**). These last two messages are sent in return to a check on the status of the party.

Channel **CB** allows the switching system to send three types of messages to the called party: check on the party's status (**checkb**), ring the phone (**ringb**), and stop ringing the phone (**stopringb**).

The first module description (**MA**) is for the calling party. It has one interaction point (**P**) that will be attached to an instance of channel **CA**. There is only one module body (**BA**) specified for the calling party. The states and transitions are the same as described in the finite state machine diagram. Note that there is a nondeterministic transition from the **Waiting** state when a **ringa** event occurs at the interaction point.

The second module description (**MB**) is for the called party. It has one interaction point (**P**) that will be attached to an instance of channel **CB**. There is only one module body (**BB**) specified for the called party. Again, the states and transitions are the same as the finite state machine diagram. The only nondeterminism is the possibility of a transition from **Onhook** to **Offhook**, or vice versa. These correspond to the calling party hanging up (or picking up) when there is no stimulus.

The last module description (**MG**) is for the agent. It has two interaction points: **PA** to connect

to an instance of a channel **CA** and **PB** to connect to an instance of a channel **CB**. There is only module body (**BG**) specified. The only nondeterminism is the possibility of a transition from **Camping** to **Checking**. This corresponds to the behavior of returning to check on a party that is busy, hoping to initiate a callback.

Three module variables are instantiated after all of the module descriptions. The initialization section creates the three module instances and connects the two channels.

3.2.3 Commentary

The declaration of message types in Estelle is helpful. It provides redundancy that helps to check for possible mistakes made in the use of those message types. It also summarizes useful information at the appropriate level. The same benefits may be claimed for the declaration of states in each process.

The rest of the syntax for Estelle is not so helpful. Pascal-style syntax may be familiar to programmers, but it is not so familiar to requirements authors. The use of indentation and keywords places too much emphasis on how the logic can be implemented, rather than on what it should be.

Most of the tools for Estelle support this bias toward implementation. There are tools for compiling Estelle to C and for simulation of Estelle specifications. But, there are few tools for analysis of Estelle specifications.

3.3 LOTOS

The Language Of Temporal Ordering Specifications (LOTOS) was developed to specify protocols. It has a process behavior part, called Basic LOTOS, that follows the process algebra paradigm, similar to CCS [Milner 1980] and CSP [Hoare 1985]. The full language also includes a part for describing data that is passed between agents. A good tutorial introduction to LOTOS may be found in [Bolognesi and Brinksma 1987].

3.3.1 Language Overview

A LOTOS specification consists of a set of process definitions. There is usually one process that defines the entire system of interest.

3.3.1.1 Structure

Unlike Estelle and SDL, LOTOS has no linguistic elements for describing the processing components and channels of a system. Instead, one adopts a convention of naming processes and data elements so that they may be distinguished as belonging to different classes of objects.

3.3.1.2 Processes

Each process is described by a single behavior expression. The basic elements of expressions are:

alternation ([])
Any of the sub-behavior operands may be selected as the behavior of the expression. This is similar to a case statement, except that the choice is made nondeterministically.

parallel composition ([])
This operator takes three arguments: two sub-behaviors and a list of events. The

events are shared between the sub-behaviors. That is, a listed event can only occur if both sub-behaviors are ready to offer that event. If only one is ready, then the other behavior must wait or execute another event (if a choice is available).

process instantiation
This is like a function call---the named process describes the behavior of the expression.

prefix (;)
An event (the first operand) must occur next, followed by the behavior of the second operand. This is similar to a sequence of statements, where the first statement is the occurrence of the event, and the second statement is the sub-behavior operand.

Vissers et el. [Vissers et al. 1991] illustrate several styles that one can use in writing LOTOS specifications. I have chosen to use the extended automata style of Basic LOTOS in this paper, as it most closely matches the styles used with the other languages.

3.3.2 Example

Figure 10 shows a fragment of the AC example in Basic LOTOS. This fragment includes all of the event declarations and parameters. Unfortunately, these event lists are repeated several times in the specification, rendering it almost unreadable. A full example is contained in Figures 11 through 14, with event declarations and parameters elided.

The beginning of the specification indicates that the system is described by an instance of one process, **root**, whose definition appears at the end. This process is the parallel composition of three other processes: **partya**, **partyb** and **agent**. The **agent** process shares events with **partya** and with **partyb**. These sharings model the behavior of passing messages between the processes.

Throughout the specification, the first part of a process name is used to show its origin. E.g., a name beginning with **partya** indicates that it describes part of the **partya** process. The complete description of a process can only be achieved by examining all of the process definitions that have that prefix. The second part of a process name represents its state. E.g., **partya_Idle** corresponds to the Idle state of the **partya** process.

To see how **partya** behaves, start with its definition at the beginning of the specification. Its behavior consists of one expression, an instance of **partya_Idle**. This models a transition to the **Idle** state. The **partya_Idle** process is described by the expression “**ac**; **partya_Waiting**”. This models the behavior of experiencing the **ac** event, followed by a transition to the **Waiting** state. From the **Waiting** state there is a **ringa** event followed by a transition to the **Ringing** state. The **partya_Ringing** process is defined as a choice between two expressions, corresponding to answering or not answering.

Figure 11 contains the complete specification of the **partya** process. Figure 12 contains the description of **partyb**. The **agent** process spans Figures 13 and 14.

3.3.3 Commentary

The process algebra paradigm of Basic LOTOS is the most abstract of all the notations reviewed here. This is both a benefit and a disadvantage. The benefit is in emphasizing the **what** rather than the **how** in requirements specifications. The disadvantage is in presenting an unusual style of notation that must be mastered. (E.g., recursive process definitions are particularly diffi-

cult to teach to naive users.)

As I observed in previous studies [Ardis 1994], some of the syntax of LOTOS is flawed. Declarations of event types are missing, thus depriving the analyst of useful redundant information that could be used to find careless mistakes. Other declarations, such as the arguments to process instantiations, are of little value in specification, but a source of unnecessary mistakes. These flaws can be remedied easily, so they may not be a significant barrier to adoption.

There are good tools for checking the consistency of LOTOS specification with temporal logic formulae (e.g., the Concurrency Workbench[Cleaveland et al. 1993]). Alternatively, it is possible to encode some such formulae as process definitions in LOTOS. This obtains much of the same value while avoiding the need to introduce yet another language and associated technology.

3.4 SDL

SDL (Specification and Description Language) was developed by the switching systems industry and was first standardized by the CCITT in 1976. Like Estelle, it is based on finite state automata, but it uses a graphical representation of flowcharts to show allowed transitions. The interconnections between processes are also described graphically. There are alternative equivalent notations that do not use pictures (and which are, therefore, more amenable to processing by analysis tools), but the graphical form is more popular.

3.4.1 Language Overview

Each process in SDL is a finite transducer, allowing the generation of outputs on each transition. Inputs correspond to messages received by the process, and outputs correspond to messages sent to other processes.

The communication topology of processes is described by a block structure that shows channels connecting blocks. Each block may be a process or may contain a substructure. Communication over the channels is asynchronous, with an unbounded queue at the receiving end of each channel.

3.4.1.1 Blocks and Channels

A system is described in SDL by a **block** diagram. At this level one identifies **channels** and subblocks, as well as specifying interactions with the surrounding environment. Subblocks may be decomposed into subblocks and channels by further block diagrams. Ultimately one reaches a level where a block is described as a set of communicating **processes**.

The **signal** language element is used to define the messages that are transmitted over a channel. The channel itself is defined by labelling an arc between two blocks. The arc may be annotated with the signals that are sent and received, or a shorthand **signallist** symbol may be used. It is also possible to specify that a channel has no delay, indicating that messages are received immediately after they are sent.

3.4.1.2 Processes

A process is defined by a process graph that usually consists of several pages of state transitions. Each page depicts a tree, with the start state at the root, and possible transitions from that state shown as branches, each branch ending with a symbol showing the new state. If there are too many possible transitions to fit on the page the tree is continued onto another page with the same start state at the top.

There are several special graphical symbols in SDL, but the most common are:

- state
- input
- output
- task

A **state** symbol is used to define a state, or to indicate the next state of a transition. An **input** symbol is used to denote receipt of a message or the firing of a timer. It may also be used with the special value none to indicate a nondeterministic choice on no input. The **output** symbol is used to show generation of an output message. A **task** symbol is used to describe computation. The description may be formal or informal. For example, one may set or reset a timer in a task.

Timers are local to processes, so they may not be used to communicate between processes. It is, of course, possible to send a message to another process that results in the setting of a timer there. But, the expiration of that timer will be local to that process. There is a primitive form of sharing of variables, but it is just a shorthand for sending messages that pass the values of local variables from one process to one another. Only the process that owns a variable may update its value.

3.4.2 Example

Figure 15 shows the system level block diagram for the AC example. The box at the top with its upper right corner folded over is a text inclusion. This style of box is used wherever text declarations are needed. This particular box includes the definitions of the **signals** (events) and the shorthand abbreviations **OutCA**, **InCA**, **OutCB**, and **InCB**. These abbreviations are used to annotate the **channels CA** and **CB**. Note that the arrows on the channels are not at the ends of the lines. This indicates that the channels are buffered. The three square boxes **PA**, **PB**, and **AG** represent the three processes PartyA, PartyB, and Agent. At this level there behavior is unknown, but there interfaces are established.

Figure 16 contains three block diagrams, one for each of the processes. Each block contains one process, denoted an octagonal polygon. For example, the process **PartyA** is a process defined for the block **PA**. The channels in these block diagrams have their arrowheads at the ends of the lines, indicating that there is no additional buffering beyond the block boundary. At the edge of each block the channel name from the system level diagram appears.

Figure 17 describes the process for **PartyA**. Three diagrams are used, numbered in the upper right corner. The first diagram starts with an empty oval, denoting the creation of the process. There is an immediate transition to the **Idle** state. The input none is used to indicate an immediate transition from **Idle** to **Waiting**. The next transition, to **Waitnext**, is taken after the **ringa** event it received. The second diagram describes transitions from the **Waitnext** state. Since both paths have a **none** input, this is a nondeterministic choice. In one case the **noanswera** message is sent as output. In the other case the **answera** message is sent. The third diagram shows transitions from the **Calling** state.

Figure 18 contains the process description for PartyB, and Figure 19 contains the process description for the Agent. Both of these figures contain a single diagram. For small process descriptions, it is convenient to draw all the transition in one diagram. However, most industrial-size systems have several diagrams per process.

3.4.3 Commentary

The flowchart-style of specification is quite natural, but it is easy to abuse. We are all familiar with flowcharts that convey little information beyond the needless complexity of the specification. One controversial point about SDL is that it allows informal text within the boxes of the flowcharts. This is claimed to be a benefit for gradual refinement of specifications. Unfortunately, it is also an excuse to postpone decisions, thus defeating one of the main purposes of requirements specification.

Temporal logic can be used to augment SDL specifications to check their accuracy. This has been done successfully at Bell Labs[Chaves 1992], though its use is not widespread in the industry. These formulae can be checked with efficient model-checking technology.

The CASE tools that support SDL are quite mature, though expensive. Most of the emphasis of these tools is on low-level design and implementation. This is probably appropriate to build an established base of experienced users, since there are more implementers than requirements writers. Eventually, though, more analysis tools will need to be added.

3.5 MSC

Message Sequence Charts (MSC) [CCITT 1992] are a graphical representation for scenarios. They evolved from common practice, and were standardized as part of the SDL work. There has been some recent interest in extending the notation so that complete algorithms could be defined by a chart, but common use is to show a single scenario per chart.

3.5.1 Language Overview

Message Sequence Charts provide a notation for describing scenarios. They are useful for specification of test cases. Each chart contains a timeline for each participating process. An arrow is drawn from one line to another whenever a message is sent between processes. Individual events that occur within a process are denoted by placing the event name next to the timeline, without an arrow.

3.5.2 Example

Figure 20 shows a scenario for the AC example.

Three processes are shown in the diagram, each in its own column. An arrow from one column to another represents a message from one process to another. An internal event is shown as a name in the column without an arrow. Time flows down the diagram.

In the example the first event is **offhookb** in the PartyB process. Next, the calling party requests automatic callback with the **ac** message. The agent sends a **checkb** message to the called party, and receives a **busyb** message in reply. The called party then goes **onhook**. The agent sends another **checkb** message, and receives the **available** message in reply this time. The agent then rings the calling party (**ringa**), and the phone is answered. The agent then rings the called party, and that phone is answered. Finally, the agent indicates that the connection has been established.

3.5.3 Commentary

Message sequence charts are the most natural style of specification for this domain. Unfortunately, they do not scale up to reasonably complete descriptions of behavior. The result is a misleading sense of security.

The notation is probably adequate for requirements, where there are not too many processes or events to describe. They are unwieldy when used for design specification. (I have seen MSCs with more than 20 process columns and more than 20 pages of events just to describe one scenario.) Thus, it is unlikely that technology will be developed to refine MSCs into designs and implementations.

One can express some temporal logic formula instances directly in MSC form. This provides the same benefits as are obtained in LOTOS: it provides a useful checking mechanism without introducing a new language.

3.6 Other Languages

Petri nets [Peterson 1977] have been used as an underlying formalism for analysis tools for several of the formal methods. They have not been used often for requirements specification. The graphical form becomes unwieldy as the size of the problem grows. Temporal logic has also been used in analysis. Unfortunately, little work has been done to make the syntax of temporal logic appealing to non-logicians.

Several languages based on extended finite state machines (e.g., Esterel[Berry and Gonthier 1992], Statecharts[Harel 1986]) are being investigated, but little practical experience has been acquired yet. Most of these languages have the disadvantage that they are not standardized by international bodies. This may seem like a simple problem in theory, but it is quite a barrier in practice.

Abstract modeling languages, such as VDM [Jones 1990] and Z [Spivey 1992], have also been tried. These languages must be extended to handle concurrency, which plays a major role in most telecommunication system specifications.

4. COMPARISON OF FORMAL LANGUAGES

In order to compare these different formal languages, it is important to first describe the criteria that will be used. The criteria I have chosen have been used before in a similar exercise with a group of experts at Bell Labs [Ardis et al. 1996].

The criteria are grouped into two categories; the order in which the criteria appear below within each group is arbitrary. The fundamental selection criteria are those that we believe are very important for a language to be successful, especially in telecommunications software development. The important selection criteria are those that we believe are generally very important, but for which we found less evidence in our setting.

The aim of these criteria is to help assess the suitability of languages for industrial software development, and they reflect concerns that arise through the various phases of the traditional waterfall model of software development. An informal assessment of the importance of each criteria during the phases of the software lifecycle is given in Figure 21. While the nature of these criteria is necessarily subjective, we believe that they accurately reflect the important considerations in our setting.

4.1 Fundamental Selection Criteria

Applicability

Can the technology describe real world problems and solutions in a natural and straightforward manner? If the technology makes assumptions about the environment, are they reasonable or benign? Is the technology compatible with the physical reality of the intended system?

Implementability: Reasonable path to code
Can the specification be easily refined or translated into an implementation that is compatible with the rest of the system? (For example, if code is generated automatically, is it in the same language/dialect as the rest of the system?) How difficult is this translation?

Testability/Simulation
Can the specification be used to test the implementation? Is it possible to execute a specification, typically by interactively providing it with input stimuli and observing that the behavior and output produced in response are as expected?

Checkability
Can a domain expert check the specification for accuracy? How readable are specifications for domain experts (as opposed to experts in the technology)?

Maintainability
Can the specification be used as the starting point for maintenance activities? Is it easy to make modifications to the specification?

Modularity
Does the method possess simple composition operators that allow a large specification to be more easily written and understood by decomposing it into smaller parts? Is it possible to make conceptually small additions, changes, and generalizations to a specification without major rewriting?

Level of abstraction/expressibility
How closely and expressively can the objects in the specification describe, from the user's point of view, the objects, equipment and the environment in the domain?

Soundness
Do the language and tools provide support for detecting inconsistencies and ambiguities in specifications? Does the language have a precisely defined semantics?

Verifiability
Is it possible to demonstrate formally that a specification or program satisfies properties stated at the same or a higher level of abstraction?

Run-time Safety
If the language supports automated code generation, does the resulting implementation degrade gracefully under unexpected run-time conditions?

Tools Maturity
How much development have the tools undergone in terms of available facilities, quality, amount and quality of training available, amount of support available (at development time, test time and run time), user base size, industrial use, momentum and impetus gained in industrial settings?

4.2 Important Selection Criteria

Looseness
Does the language permit incompleteness or non-determinism in the specifications?

Learning Curve
Can a completely new user quickly learn the concepts, techniques and heuristics to make useful application of the language?

Language Maturity
How much development has the language undergone in terms of certification, amount and quality of training available, user base size, industrial use, momentum and impetus gained in industrial settings?

Data Modeling
Does the language provide facilities to describe data representation, relationships and/or abstractions? Does it provide them in a natural, integrated way?

Discipline
Does the language force users to write reasonably well-behaved programs?

4.3 Evaluation

Using these criteria, Figure 22 shows my subjective evaluation of the formalisms described here. Some of the judgements are based on considerable experience with these languages by myself and others at Bell Labs. Unfortunately, we have not had much experience with Estelle, so that portion of the evaluation is weak.

5. RESULTS AND OPPORTUNITIES

In spite of the advantages of formal methods, they are not common practice in industry. Our group at Bell Labs has investigated the applicability of many of these methods [Ardis et al. 1996] in order to better understand this phenomenon.

5.1 Contributions Made

The first contribution to acknowledge in the use of any of these methods is precision. Most projects that use formal methods claim that they discover a large number of ambiguities and inconsistencies in informal requirements. The act of formalizing forces one to resolve these issues.

A second contribution is the ability to execute formal models over a large subset of possible scenarios. Formal specifications can be checked for deadlocks and unreachable states with model-checking algorithms [Holzmann 1992] that are quite efficient. These analyses may not be completely exhaustive, but they approach limits that are sufficient for most applications.

A third contribution of formal methods is the ability to translate from one to another. Once a problem has been formalized it can be transformed into other representations by algorithms that are (almost) guaranteed to perform flawlessly. Even methods that appear to be far removed can be shown to be equivalent. The SPECS project [Dauphin et al. 1993] has constructed compilers that

translate SDL and LOTOS into common internal forms.

A fourth contribution obtained by some of these methods is integration with traditional informal methods. SDL is now supported by commercial toolsets that offer data dictionaries and reports similar to traditional CASE tools. It is likely that the other languages will soon have similar support.

5.2 Remaining Problems

Here are some problems that prevent the widespread use of formal methods to describe requirements for telecommunications systems:

Abstraction
Although there are many abstraction mechanisms available, some important ones are still missing. In particular, events need to be collected into data abstractions. The only data abstraction mechanisms available today are for the values of data exchanged in protocols, not for the collection of abstract events.

Domain specificity
Given the rich tradition of telecommunications, it is surprising that methods are not more specialized for this domain.

User friendliness
We are unlikely to see customers express their requirements for systems using formal methods.

Much of the work in formal methods for telecommunication systems has focussed on the complexity of the algorithms and of the interaction of processes. This is similar to the progress in programming languages, where the initial focus was on structured programming to tame complex control structures. Just as programming languages turned toward data abstraction, formal methods for telecommunications must address the problems of complexity of objects.

Modern protocols have large numbers of events. It is too easy to mistype one and accidentally create another. Programming languages solved this problem by requiring declarations of variables.

Although one sees the same events in protocol after protocol (e.g., offhook, onhook), these terms have not been standardized In fact, the proponents of FDTs argue that their languages are general enough to be useful for any reactive system. But, that is also why those methods are too weak for their intended use.

The same argument holds for features that have become standard practice, but not specified standards. We need models of these features that recognize the important abstractions and their alternative definitions. The work of Zave et al. [Mataga and Zave 1995] is a step in the right direction.

Finally, we need an alternative to temporal logic that naive users will find comfortable. Those who have attempted to apply formal methods to industrial problems have found creative ways to hide predicate calculus. Tables, for example, are a convenient way to hide conjunction and disjunction, and they are easier to check for completeness.

6. CONCLUSION

The telecommunications industry has developed its own set of formal languages to describe requirements, and it has standardized those languages to promote their use. These languages should improve the state of requirements specifications in the industry. There is still room for improvement of these languages, however. New abstraction mechanisms, friendlier syntax, and greater standardization of the common abstractions in this domain should be added.

7. REFERENCES

[Ardis 1994]
Ardis, M. (1994), "Lessons from using Basic LOTOS," In Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, pp. 5-14.

[Ardis et al. 1996]
Ardis, M., J.A. Chaves, L.J. Jagadeesan, P. Mataga, C. Puchol, M. Staskauskas, and J. VonOlnhausen (1996), "A framework for evaluating specification methods for reactive systems," IEEE Transactions on Software Engineering 22, 6, 378-389.

[Aho and Ullman 1972]
Aho, A.V. and J.D. Ullman (1972), The Theory of Parsing, Translation, and Compiling, Prentice-Hall.

[Bellcore 1991]
Bellcore (1991), "Synchronous optical network (SONET) transport systems: Common generic criteria," Technical Report TR-NWT-000253, Issue 2, Bellcore.

[Berry and Gonthier 1992]
Berry, G. and G. Gonthier (1992), "The ESTEREL synchronous programming language: design, semantics, implementation," Science of Computer Programming 19, 87-152.

[Bolognesi and Brinksma 1987]
Bolognesi, T. and E. Brinksma (1987), "Introduction to the ISO specification language LOTOS," Computer Networks and ISDN Systems 14, 25-59.

[CCITT 1992]
CCITT (1992), "Recommendation Z.120: Message sequence chart (MSC)" .

[Chaves 1992]
Chaves, J.A., "Formal methods at AT&T -- an industrial usage report," In Proceedings of Formal Description Techniques IV, North-Holland, Amsterdam, pp. 83-90.

[Cleaveland et al. 1993]
Cleaveland, R., J. Parrow, and B. Steffen (1993), "The concurrency workbench: A semantics-based tool for the verification of finite-state systems," ACM Transactions on Programming Languages and Systems 15, 1, 36-72.

[Dauphin et al. 1993]
Dauphin, M., G. Fonade, and R. Reed (1993), "SPECS: Making formal techniques usable," IEEE Software 10, 6, 55-57.

[Harel 1986]

Harel, D. (1986), "Statecharts, a visual formalism for complex systems," Technical report, The Weizmann Institute of Science.

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

[Holzmann 1992]
Holzmann, G.J. (1992), "Practical methods for the formal validation of SDL specifications," Computer Communications, pp. 129-134 (March).

[ISO 1989a]
ISO (1989), "ESTELLE---A Formal Description Technique Based on an Extended State Transition Model," International Standard ISO 9074.

[ISO 1989b]
ISO (1989), "LOTOS---A Formal Description Technique Based on the Temporal Ordering of Observational Behavior," International Standard ISO 8807.

[ITU 1993]
ITU-T (1993), "Specification and Description Language SDL.," Recommendation Z.100.

[Jones 1990]
Jones, C.B. (1990), Systematic Software Construction Using VDM, Prentice-Hall.

[Milner 1980]
Milner, R. (1980), A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science, Springer-Verlag.

[Mataga and Zave 1995]
Mataga, P.A. and P. Zave (1995), "Multiparadigm specification of an AT&T switching system," In Applications of Formal Methods, M.G. Hinchey and J.P. Bowen, editors, Prentice-Hall International.

[Peterson 1977]
Peterson, J.L. (1977), "Petri nets,"ACM Computing Surveys 9, 3, 223-252.

[Spivey 1992]
Spivey, J.M. (1992), The Z Notation: A Reference Manual, Prentice Hall International, 2nd edition.

[Vissers et al. 1991]
Vissers, C.A., G.Scollo, M.van Sinderen, and E.Brinksma (1991), "Specification styles in distributed systems design and verification," Theoretical Computer Science, 179-206.

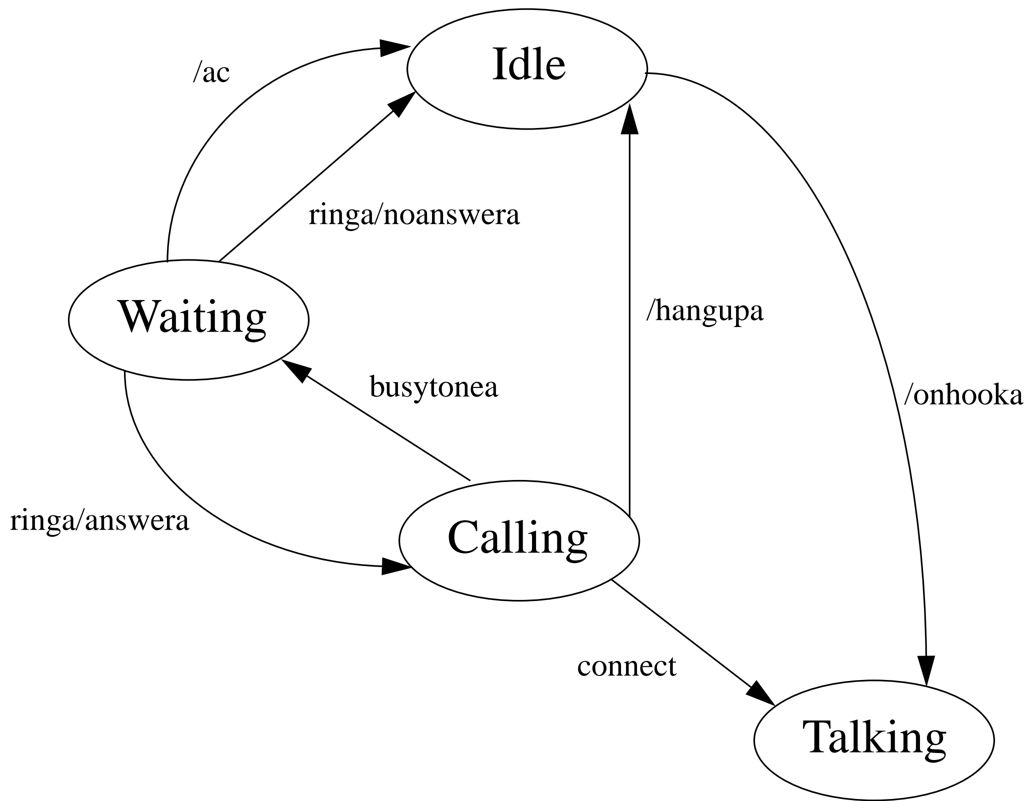


Figure 1: Calling party finite state machine diagram

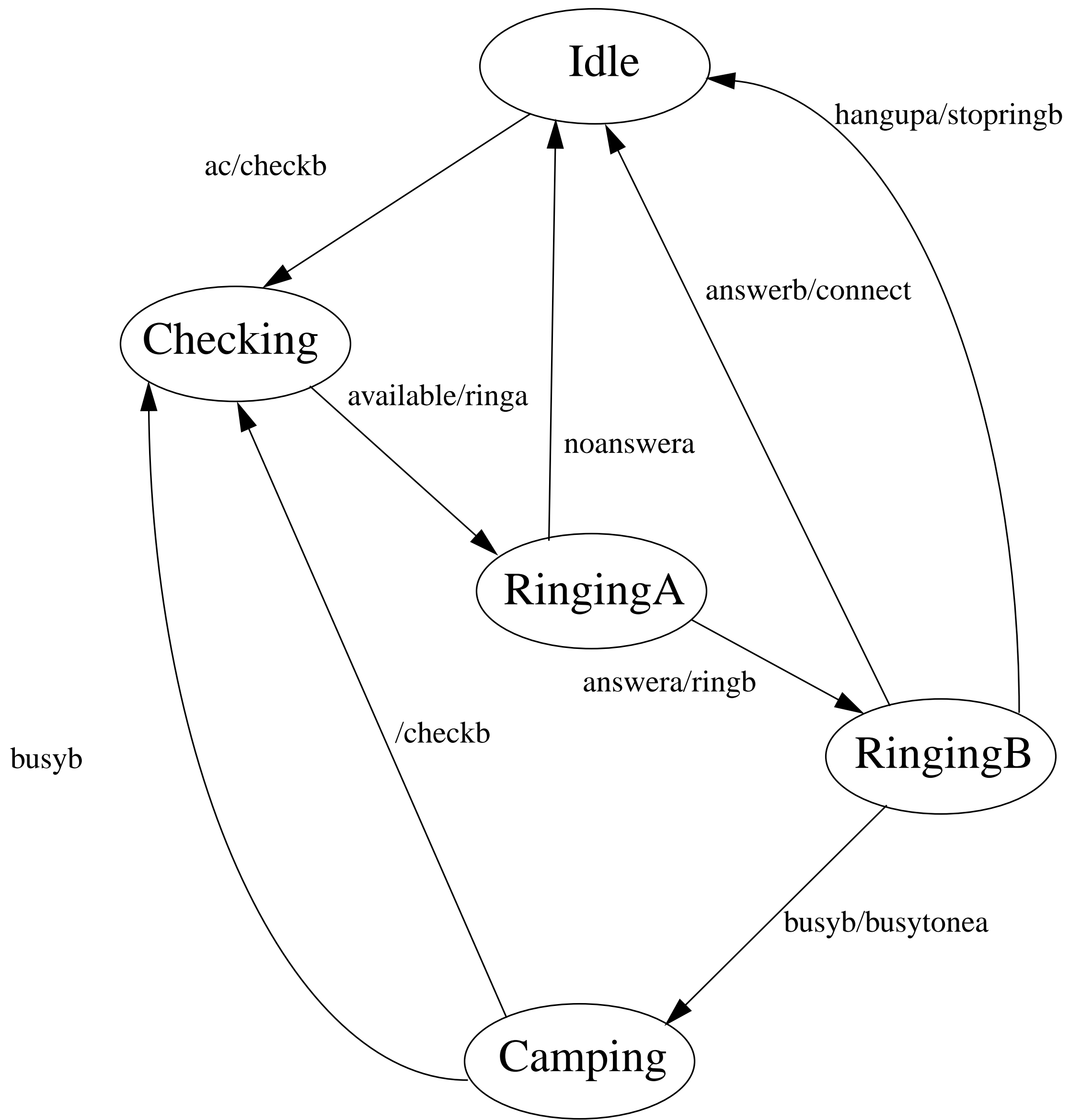


Figure 2: Switching system agent finite state machine diagram

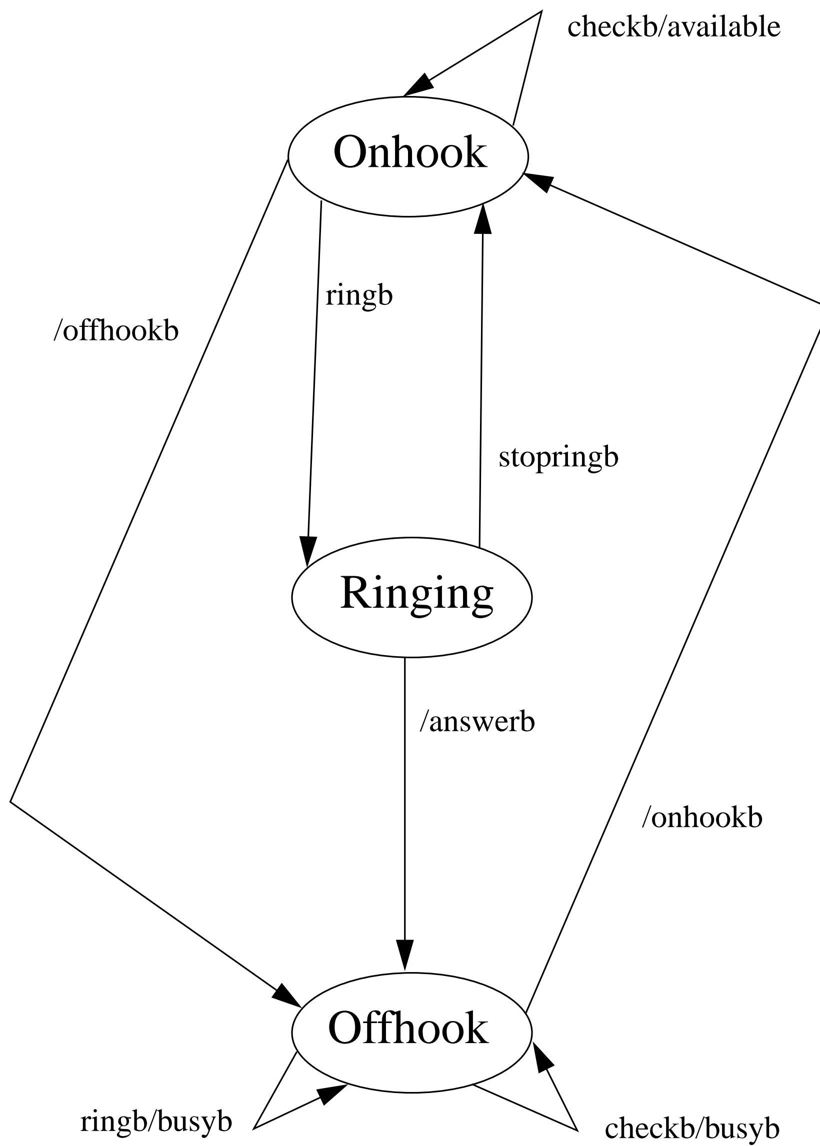


Figure 3: Called party finite state machine diagram

	RMW	RMP	RSW	RSP	FSW	FSP	CSW	CSP	DGW	DGP	FLW	FLP	CLW	CLP
WNN	PON	WNO	—	—	—	PNN	—	PNN	PDN	WND	—	—	—	—
WDN	PON	WDO	—	—	—	PDN	—	PDN	—	WDD	PFN	—	WNN	—
WFN	PON	WFO	—	—	—	PFN	—	PFN	—	WFD	—	—	WNN	—
WND	POD	WNO	—	—	—	PND	—	—	WDD	—	—	WNF	—	WNN
WDD	POD	WDO	—	—	—	PDD	—	PDD	—	—	PFD	WDF	WND	PDN
WFD	POD	WFO	—	—	—	PFD	—	PFD	—	—	—	WFF	WND	PFN
WNF	POF	WNO	—	—	—	PNF	—	—	WDF	—	—	—	—	WNN
WDF	POF	WDO	—	—	—	PDF	—	—	—	—	WFF	—	WNF	PDN
WFF	POF	WFO	—	—	—	PFF	—	PFF	—	—	—	—	WNF	PFN
WNO	WOO	—	—	WNN	—	—	—	—	WDO	—	—	—	—	—
WDO	WOO	—	—	PDN	—	—	—	—	—	—	WFO	—	WNO	—
WFO	WOO	—	—	PFN	—	—	—	—	—	—	—	—	WNO	—
WOO	—	—	WNO	PON	—	—	—	—	—	—	—	—	—	—
PNN	PON	WNO	—	—	WNN	—	WNN	—	PDN	WND	—	—	—	—
PDN	PON	WDO	—	—	WDN	—	—	—	—	PDD	PFN	—	PNN	—
PFN	PON	WFO	—	—	WFN	—	—	—	—	PFD	—	—	PNN	—
PON	—	POO	PNN	—	—	—	—	—	—	POD	—	—	—	—
PND	POD	WNO	—	—	WND	—	WND	—	PDD	—	—	WNF	—	PNN
PDD	POD	WDO	—	—	WDD	—	WDD	—	—	—	PFD	WDF	WND	PDN
PFD	POD	WFO	—	—	WFD	—	—	—	—	—	—	PFF	WND	PFN
POD	—	POO	WND	—	—	—	—	—	—	—	—	POF	—	PON
PNF	POF	WNO	—	—	WNF	—	WNF	—	PDF	—	—	—	—	PNN
PDF	POF	WDO	—	—	WDF	—	WDF	—	—	—	PFF	—	WNF	PDN
PFF	POF	WFO	—	—	WFF	—	WFF	—	—	—	—	—	WNF	PFN
POF	—	POO	WNF	—	—	—	—	—	—	—	—	—	—	PON
POO	—	—	WNO	PON	—	—	—	—	—	—	—	—	—	—

Figure 4: A table specification of Automatic Protection Switching

```
channel Buy(Customer, Vendor);
    by Customer:
    coin;
    by Vendor:
    trinket;

p1: Buy(Customer);
p2: Buy(Vendor);
```

Figure 5: Example channel type and instances in Estelle

```
specification AC;
channel CA (PartyA, Agent);
  by PartyA: ac; answera; hangupa; noanswera;
  by Agent: busytonea; connect; ringa;

channel CB (PartyB, Agent);
  by PartyB: answerb; available; busyb;
  by Agent: checkb; ringb; stopringb;

module MA systemprocess;
  ip P: CA(PartyA);
end;

body BA for MA;
  state Idle, Waiting, Calling, Talking;
  initialize to Idle;
  trans from Idle to Waiting
    begin output P.ac end;
  trans when P.ringa
    from Waiting to Calling
      begin output P.answera end;
    from Waiting to Idle
      begin output P.noanswera end;
  trans when P.busytonea
    from Calling to Waiting
      begin end;
  trans when P.connect
    from Calling to Talking
      begin end;
  trans from Calling to Idle
    begin output P.hangupa end;
  trans from Talking to Idle
    begin output P.onhooka end;
end;
```

Figure 6: Estelle specification of Automatic Callback (Part 1 of 4)

```
module MB systemprocess;
  ip P: CB(PartyB);
end;

body BB for MB;
  state Offhook, Onhook, Ringing;
  initialize to Onhook;
  trans when P.checkb
    from Offhook to Offhook
      begin output P.busyb end;
    from Onhook to Onhook
      begin output P.available end;
  trans when P.ringb
    from Onhook to Ringing
      begin end;
    from Offhook to Offhook
      begin output P.busyb end;
  trans when P.stopringb
    from Ringing to Onhook
      begin end;
  trans from Ringing to Offhook
    begin output P.answerb end;
  trans from Onhook to Offhook
    begin output P.offhookb end;
  trans from Offhook to Onhook
    begin output P.onhookb end;
end;
```

Figure 7: Estelle specification of Automatic Callback (Part 2 of 4)

```
module MG systemprocess;
 ip PA: CA(Agent);
    PB: CB(Agent);
end;

body BG for MG;
 state Idle, Checking, RingingA, RingingB, Camping;
 initialize to Idle;
 trans when PA.ac
  from Idle to Checking
   begin output PB.checkb end;
 trans when PB.busyb
  from Checking to Camping
   begin end;
  from RingingB to Camping
   begin output PA.busytonea end;
 trans when PB.available
  from Checking to RingingA
   begin output PA.ringa end;
 trans when PA.answera
  from RingingA to RingingB
   begin output PB.ringb end;
 trans when PA.noanswera
  from RingingA to Idle
   begin end;
 trans when PB.answerb
  from RingingB to Idle
   begin output PA.connect end;
 trans when PA.hangupa
  from RingingB to Idle
   begin output PB.stopringb end;
 trans from Camping to Checking
  begin output PB.checkb end;
end;
```

Figure 8: Estelle specification of Automatic Callback (Part 3 of 4)

```
modvar  PartyA: MA; PartyB: MB; Agent: MG;

initialize
  begin
    init PartyA with MA;
    init PartyB with MB;
    init Agent with MG;
    connect PartyA.P to Agent.PA;
    connect PartyB.P to Agent.PB;
  end;
end.
```

Figure 9: Estelle specification of Automatic Callback (Part 4 of 4)

```
SPECIFICATION ACSystem[ac, answera, answerb, available, busyb, busytonea,
   checkb, connect, hangupa, noanswera, offhookb, onhooka,
   onhookb, ringa, ringb, stopringb]: NOEXIT

BEHAVIOUR
  root [ac, answera, answerb, available, busyb, busytonea,
       checkb, connect, hangupa, noanswera, offhookb, onhooka,
       onhookb, ringa, ringb, stopringb]
WHERE

PROCESS partya[ac, answera, busytonea, connect, hangupa, noanswera,
             onhooka, ringa]: NOEXIT :=
  partya_Idle [ac, answera, busytonea, connect, hangupa, noanswera,
             onhooka, ringa]
ENDPROC
PROCESS partya_Calling[ac, answera, busytonea, connect, hangupa,
                  noanswera, onhooka, ringa]: NOEXIT :=
  (hangupa;
  partya_Idle[ac, answera, busytonea, connect, hangupa, noanswera,
            onhooka, ringa]
  []
  busytonea;
  partya_Waiting[ac, answera, busytonea, connect, hangupa, noanswera,
              onhooka, ringa]
  []
  connect;
  partya_Talking[ac, answera, busytonea, connect, hangupa, noanswera,
              onhooka, ringa])
ENDPROC
...
```

Figure 10: Fragment of LOTOS specification

```
SPECIFICATION ACSystem[..]  : NOEXIT
BEHAVIOUR
  root [...]
WHERE
PROCESS partya[...]: NOEXIT :=
  partya_Idle [...]
ENDPROC

PROCESS partya_Calling[...]: NOEXIT :=
  (hangupa; partya_Idle [...]
  []
  busytonea; partya_Waiting[...]
  []
  connect; partya_Talking[...])
ENDPROC

PROCESS partya_Idle[...]: NOEXIT :=
  (ac; partya_Waiting [...])
ENDPROC

PROCESS partya_Ringing[...]: NOEXIT :=
  (answera; partya_Calling [...]
  []
  noanswera; partya_Idle [...])
ENDPROC

PROCESS partya_Talking[...]: NOEXIT :=
  (onhooka; partya_Idle [...])
ENDPROC

PROCESS partya_Waiting[...]: NOEXIT :=
  (ringa; partya_Ringing [...])
ENDPROC
```

Figure 11: LOTOS specification of Automatic Callback (Part 1 of 4)

```
PROCESS partyb[...]: NOEXIT :=
  partyb_Onhook[...]
ENDPROC

PROCESS partyb_Avail[...]: NOEXIT :=
  (available; partyb_Onhook[...])
ENDPROC

PROCESS partyb_Busy[...]: NOEXIT :=
  (busyb; partyb_Offhook[...])
ENDPROC

PROCESS partyb_Offhook[...]: NOEXIT :=
  (onhookb; partyb_Onhook[...]
  []
  ringb; partyb_Busy[...]
  []
  checkb; partyb_Busy[...])
ENDPROC

PROCESS partyb_Onhook[...]: NOEXIT :=
  (offhookb; partyb_Offhook[...]
  []
  ringb; partyb_Ringing[...]
  []
  checkb; partyb_Avail[...])
ENDPROC

PROCESS partyb_Ringing[...]: NOEXIT :=
  (answerb; partyb_Offhook [...]
  []
  stopringb; partyb_Onhook [...])
ENDPROC
```

Figure 12: LOTOS specification of Automatic Callback (Part 2 of 4)

```
PROCESS agent[...]: NOEXIT :=
  agent_Idle [...]
ENDPROC

PROCESS agent_Alerta[...]: NOEXIT :=
  (ringa; agent_Ringinga [...])
ENDPROC

PROCESS agent_Alertb[...]: NOEXIT :=
  (ringb; agent_Ringingb [...])
ENDPROC

PROCESS agent_Busying[...]: NOEXIT :=
  (busytonea; agent_Camping [...])
ENDPROC

PROCESS agent_Camping[...]: NOEXIT :=
  (checkb; agent_Checking [...])
ENDPROC

PROCESS agent_Checking[...]: NOEXIT :=
  (busyb; agent_Camping [...]
  []
  available; agent_Alerta [...])
ENDPROC

PROCESS agent_Connecting[...]: NOEXIT :=
  (connect; agent_Idle [...])
ENDPROC

PROCESS agent_Idle[...]: NOEXIT :=
  (ac; agent_Request [...])
ENDPROC
```

Figure 13: LOTOS specification of Automatic Callback (Part 3 of 4)

```
PROCESS agent_Quitting[...]: NOEXIT :=
  (stopringb; agent_Idle [...])
ENDPROC

PROCESS agent_Request[...]: NOEXIT :=
  (checkb; agent_Checking [...])
ENDPROC

PROCESS agent_Ringinga[...]: NOEXIT :=
  (answera; agent_Alertb [...]
  []
  noanswera; agent_Idle [...])
ENDPROC

PROCESS agent_Ringingb[...]: NOEXIT :=
  (answerb; agent_Connecting [...]
  []
  busyb; agent_Busying [...]
  []
  hangupa; agent_Quitting [...])
ENDPROC

PROCESS root[...]: NOEXIT :=
  ((partya [...]
  |[  ]|
  partyb [...])
  |[ ... ]|
  agent [...])
ENDPROC

ENDSPEC
```

Figure 14: LOTOS specification of Automatic Callback (Part 4 of 4)

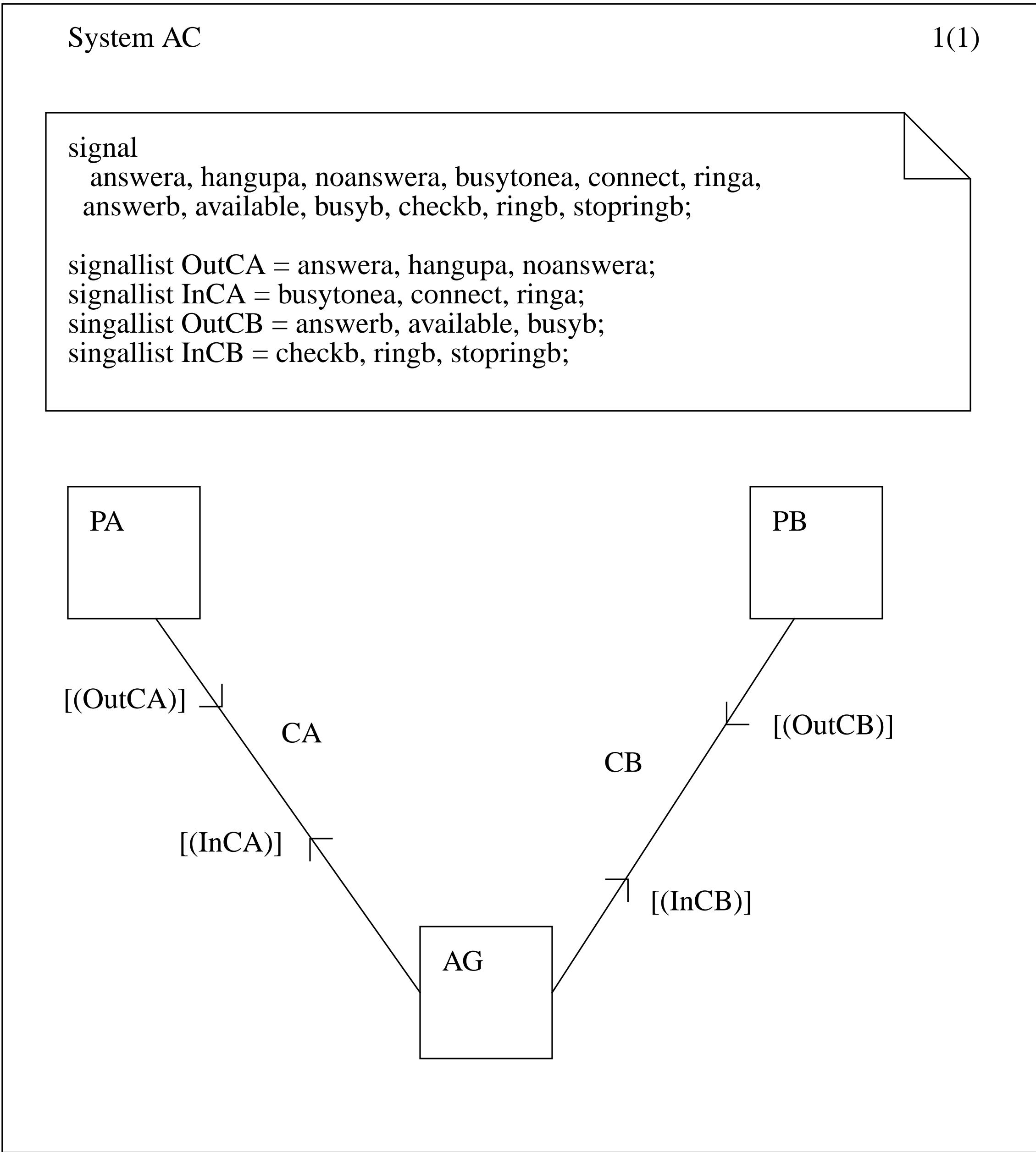


Figure 15: SDL system level block diagram

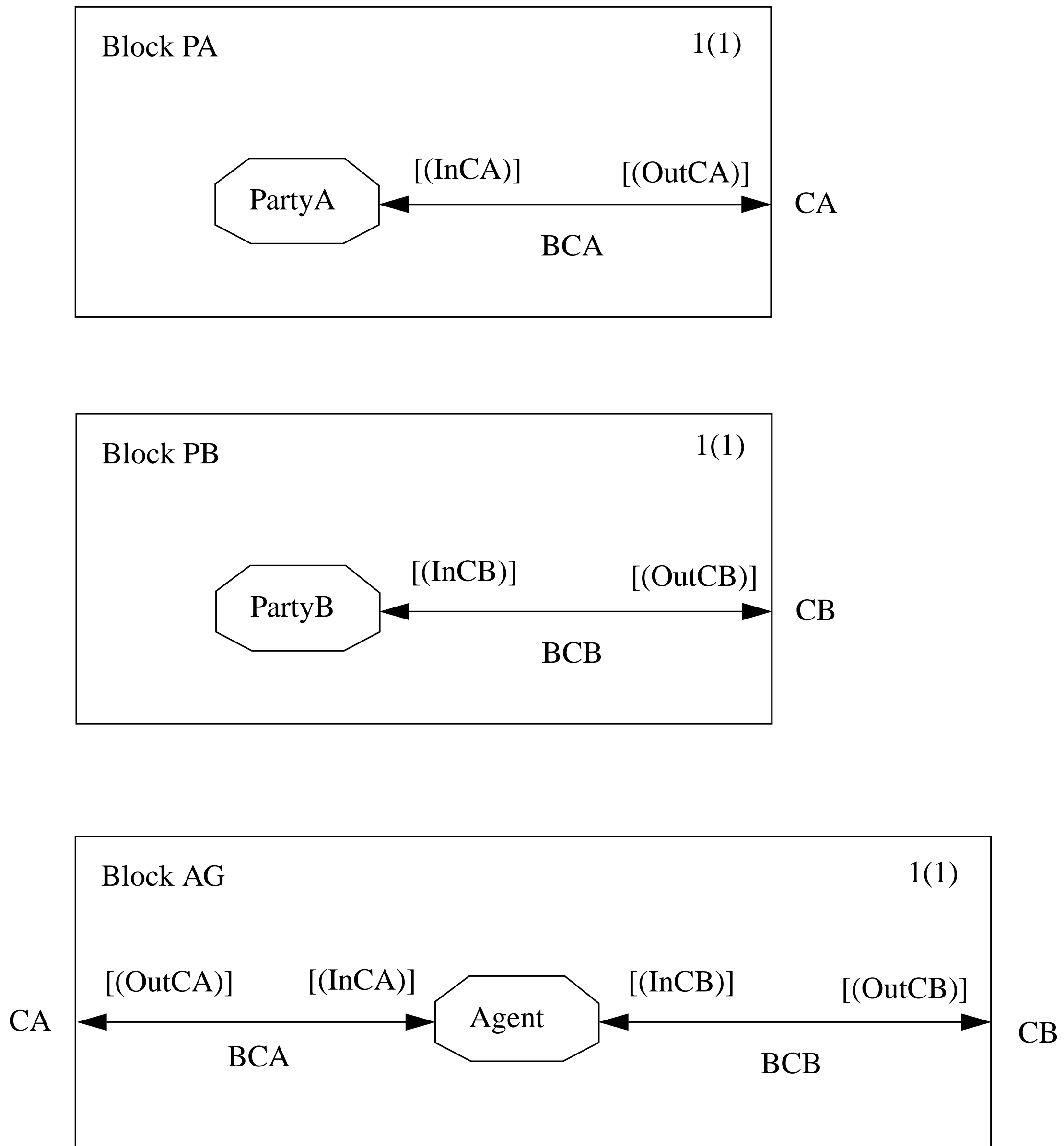
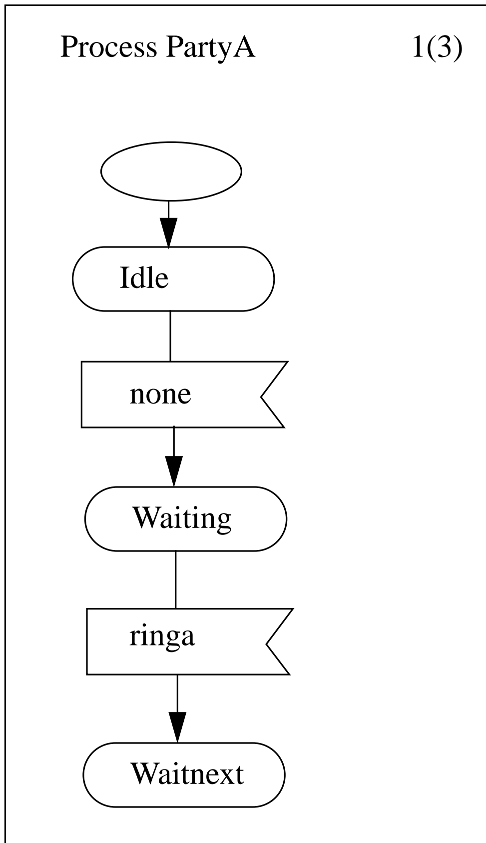
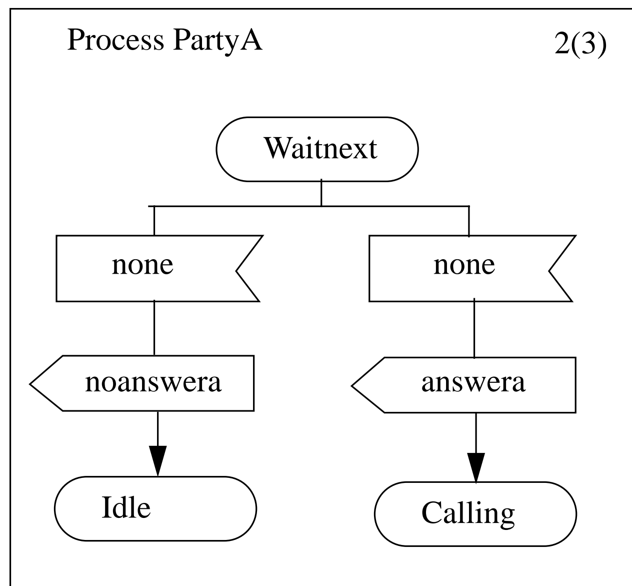


Figure 16: SDL block diagrams





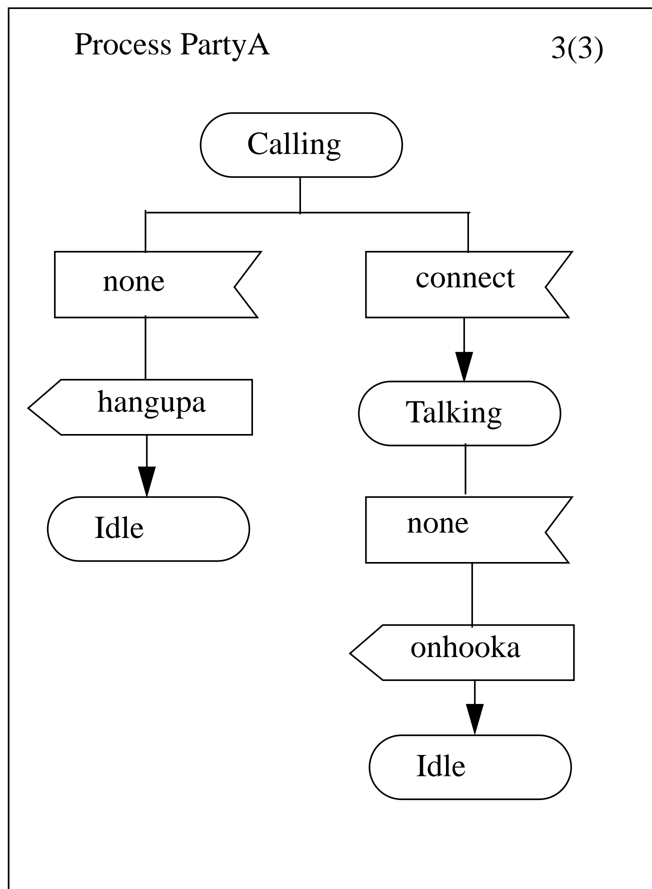


Figure 17: SDL process for PartyA

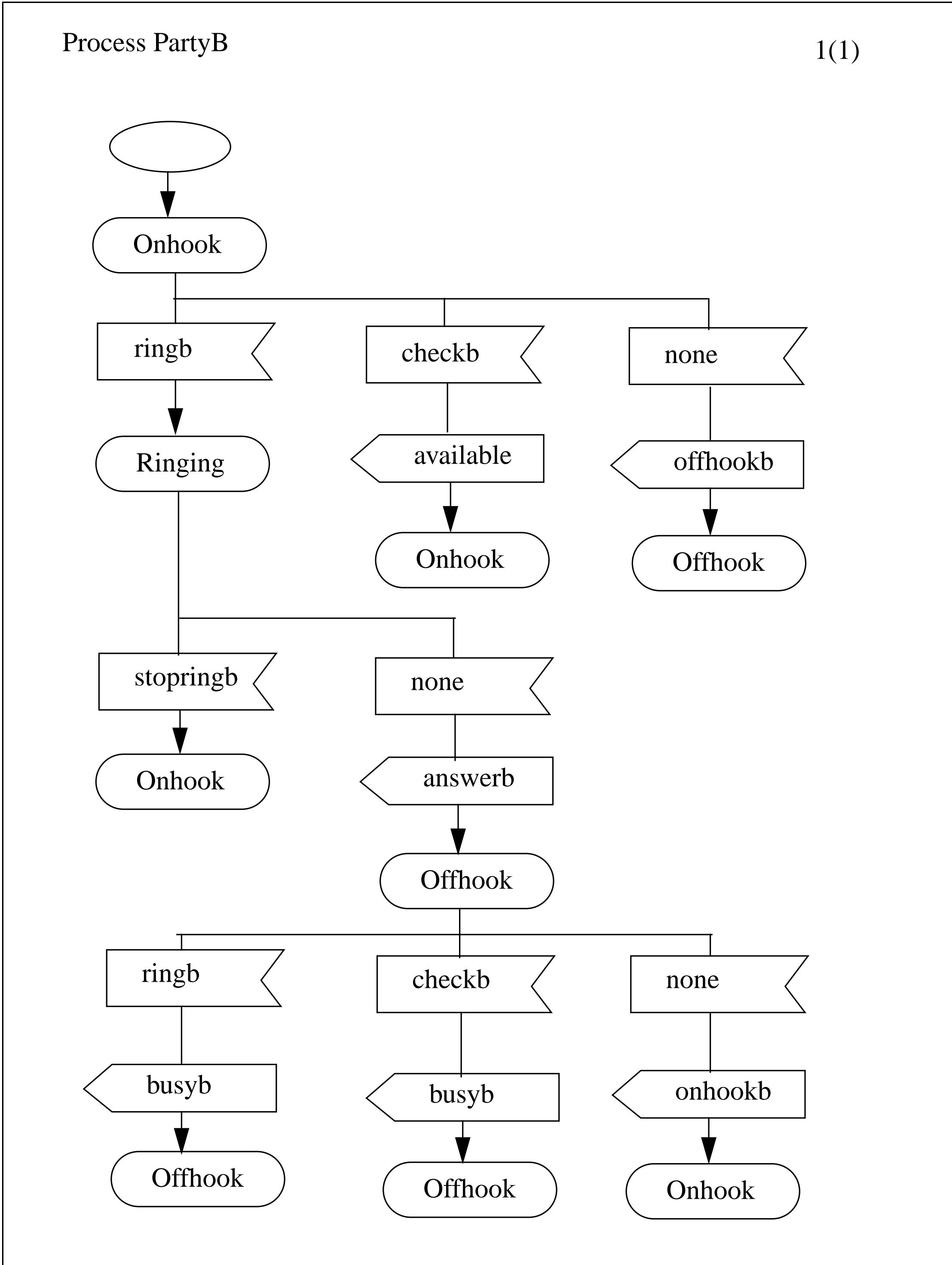


Figure 18: SDL process PartyB

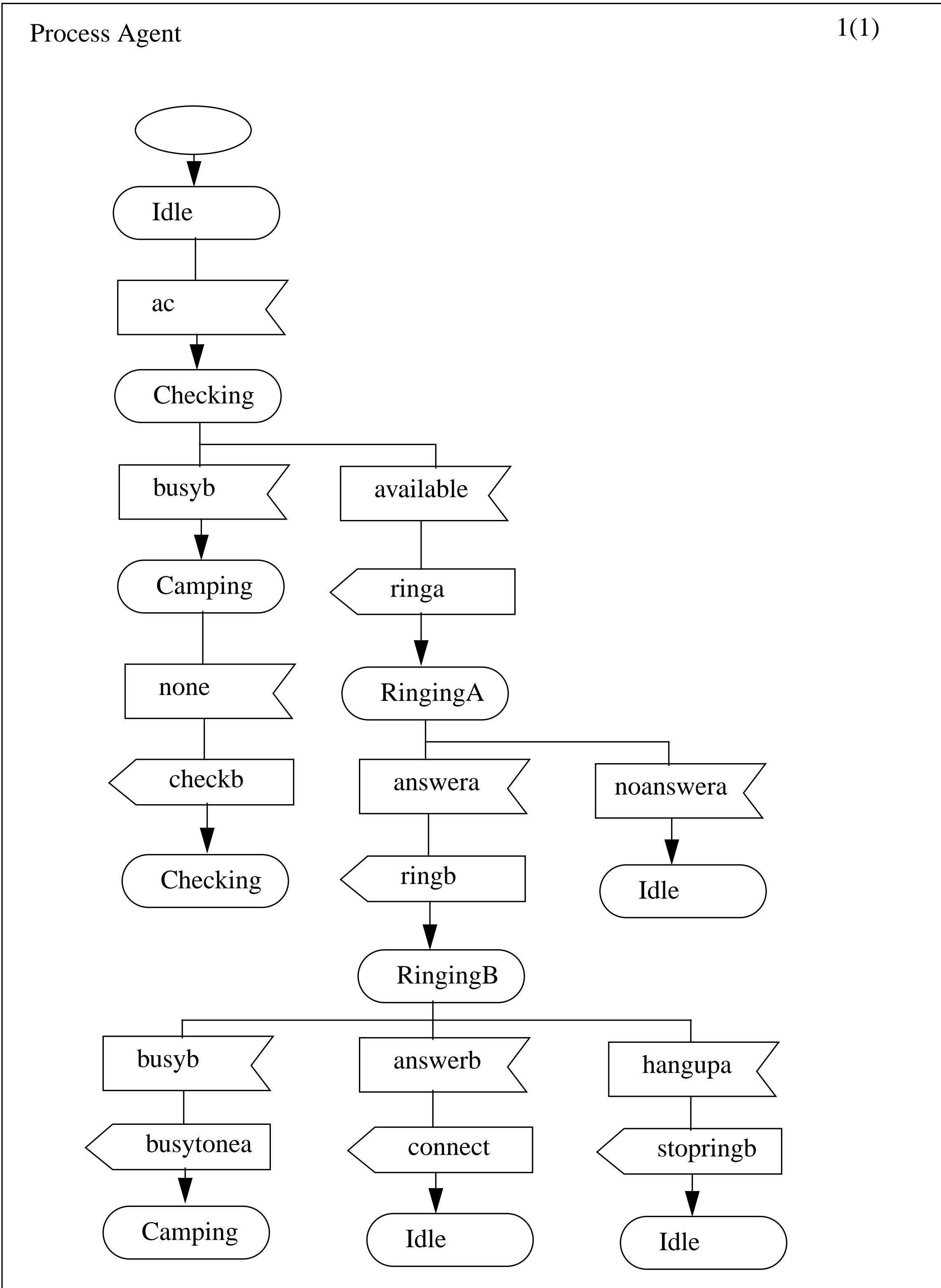


Figure 19: SDL process Agent

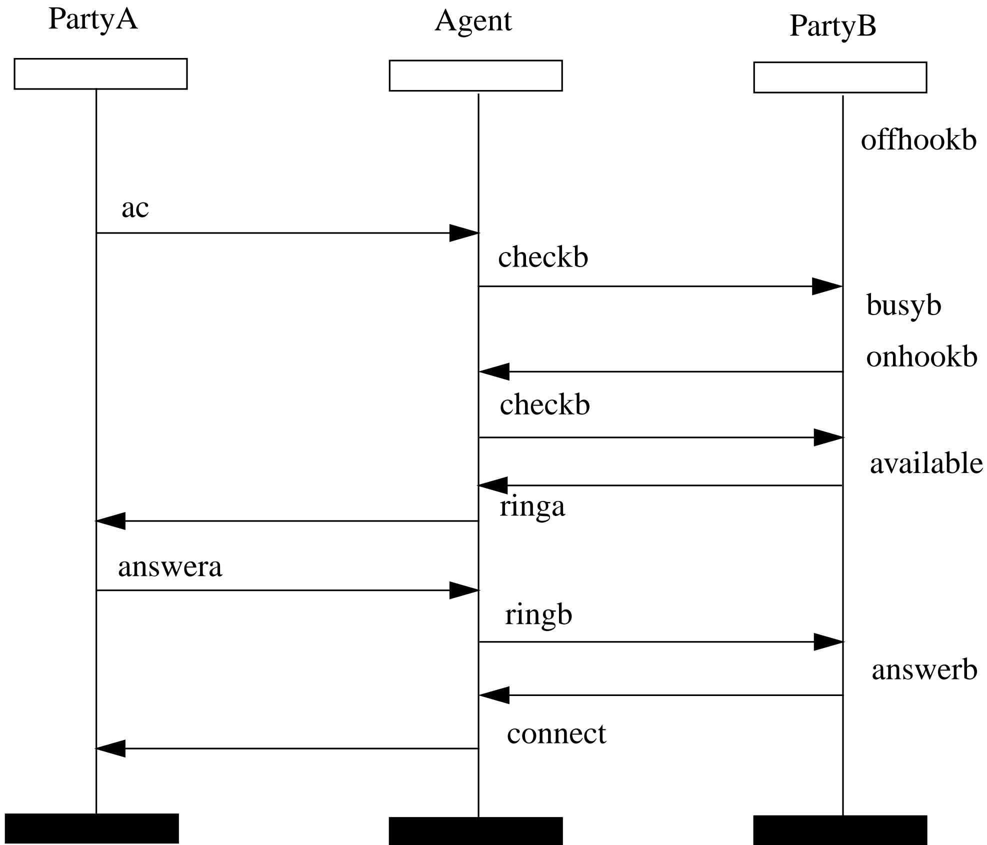


Figure 20: MSC example

Phases						Criteria
R	D	I	T	M	O	
+		+		+		Applicability
		+		+		Implementability
+	+		+			Testability/Simulation
+			+	+		Checkability
				+		Maintainability
	+			+		Modularity
+	+					Level of abstraction
+			+			Soundness
+	+	+	+	+		Verifiability
		+		+		Run-time Safety
		+	+	+		Tool Maturity
+						Looseness
					+	Learning Curve
					+	Language Maturity
	+					Data Modeling
+	+	+		+		Discipline

R = requirements phase

D = design phase

I = implementation phase

T = test phase

M = maintenance phase

O = Other (does not fit any particular phase)

+ = Criterion is important during phase

Figure 21: Importance of evaluation criteria throughout software lifecycle

Criteria	**Estelle**	**LOTOS**	**SDL**
Applicability	+	+	+
Implementability	0	0	+
Testability/Simulation	+	+	+
Checkability	0	0	+
Maintainability	0	-	0
Modularity	0	0	0
Level of abstraction	-	0	0
Soundness	0	+	0
Verifiability	0	+	+
Run-time Safety	0	0	0
Tools Maturity	-	0	+
Looseness	0	0	0
Learning Curve	0	0	+
Language Maturity	0	+	+
Data Modeling	0	0	0
Discipline	0	0	0

+ = Strength

0 = Adequate

- = Weakness

Figure 22: Comparison of languages