

Lessons from Using Basic LOTOS

Experience Report

Mark A. Ardis

Software Production Research

AT&T Bell Laboratories

Naperville, IL 60566-7013

Abstract

We describe three case studies in the use of Basic LOTOS. The studies cover design recovery, requirements specification, and design activities. We also report lessons learned from the studies. Early lessons suggested changes to the syntax of the language used, and the need for some specific analysis tools. The last case study reports some of the results of these changes.

KEYWORDS: formal specification, LOTOS, software design, software requirements, switching systems

1 Introduction

Formal methods have been proposed as an improvement over current practice in software development. In particular, formal methods claim to reduce errors during the early phases of specification and design. We sought to test these claims through the introduction of formal methods into the development of software for a large electronic switching system, the 5ESS[®].

This paper describes the results of three case studies in the use of Basic LOTOS^[15]. Each study was conducted as a joint project between research and development departments of AT&T Bell Laboratories.

2 Background

2.1 Software Development Methods

The software development process for the 5ESS follows the waterfall model, with identifiable phases of requirements definition, design, coding, and testing^[11]. The product is very large, on the order of several million lines of code, with teams of hundreds of engineers responsible for each release^[20]. Naturally, any method that promises to reduce effort or improve quality could be important to the future of the 5ESS product.

One of our concerns in introducing change in the development process was acceptance by engineers. Another concern was the difficulty of making any change in a process involving so many people. We hoped to minimize both of these problems by adopting a strategy that would require as little change in behavior as

necessary. In particular, we wanted to make it possible for only part of a team (e.g., only one person) to adopt a new method. This meant that the new method would have to be compatible with the old methods, and that some benefits would still accrue from the limited use.

Requirements and high-level designs in our areas of study were typically expressed in scenario form. That is, required behaviors of the software and hardware were specified by sequences of events in structured English. While this language of events is shared by large groups of developers, it is an informal language that has evolved over several years. We speculated that significant benefits could accrue from formalization of scenarios. Indeed, others had tried this before^[9], with promising results.

2.2 Why Basic LOTOS?

Many formal methods have been proposed for the specification of systems. We selected LOTOS for the following reasons:

- LOTOS was designed for protocol specification, and it has already been used to describe many telecommunications examples^[6]. It may become a popular language for describing switching systems.
- Different styles may be used in writing LOTOS. Of particular interest to us was the constraint-oriented style, which allows one to write several independent constraints that may be easily composed to define the complete system.
- The software we studied was already divided into processes. LOTOS processes looked much like pseudo-code descriptions of the software.
- LOTOS is easily interpreted. We felt that execution or animation of specifications would be important to developers.

Given that LOTOS was our choice, why did we restrict ourselves to Basic LOTOS, instead of the whole

language? We avoided the data definition part of LOTOS because:

- Most, if not all, of the behavior we wanted to describe could be expressed in terms of atomic events. Few events conveyed more information than their occurrence.
- We hoped to refine our specifications into a finite state machine form. For example, we wanted to test our specifications by model checking. One approach we considered was to translate from LOTOS to a dialect of CCS^[16] accepted by the concurrency workbench^[10]. Although we have not yet used this method to analyze our examples, we have already begun to investigate development methods that refine LOTOS into finite state machine formalisms.
- The data definition facilities of LOTOS are based on an algebraic specification method that is unfamiliar to most of our developers. We felt that it would be a better strategy to investigate Basic LOTOS first, since it was simpler than the whole language.
- We started with few tools, and we expected to develop our own tools as we conducted our case studies. We pursued the same strategy as other researchers in building tools for Basic LOTOS before tackling the whole language.

In the rest of the paper the term “LOTOS” really refers to “Basic LOTOS”. None of our studies considered use of the full language, though we intend to do so in the future.

2.3 What is Basic 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 and CSP^[14]. As noted above, the full language also includes a part for describing data that is passed between agents, but we did not use this part of the language in our work. A good tutorial introduction to LOTOS may be found in^[5].

A LOTOS specification consists of a set of process definitions. There is usually one process that defines the entire system of interest. Each process is described by a single behavior expression. There are several different forms of behavior expressions, such as:

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 ($[\] \mid$) 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.

Appendix A contains an example of a vending machine (adapted from ^[14]) in Basic LOTOS. Please note that this example is not written in typical LOTOS style. It is written so that the other examples that follow may be easily compared.

2.4 How is Basic LOTOS Used?

As Turner described in ^[17], a LOTOS specification can be developed by stepwise refinement. A constraint-oriented style might be adopted first, to express the requirements (the constraints) without biasing the implementation. The specification is then refined and elaborated, perhaps transforming the style of specification into the state-oriented style. A discussion of several different styles of specification in LOTOS appears in ^[19].

Because LOTOS is a formal language, it is possible to check specifications for internal consistency. Also, Basic LOTOS has an operational semantics that promotes execution or simulation. In our view, designing with Basic LOTOS can be viewed as similar in nature to programming: the author of a Basic LOTOS specification can get the same type of immediate feedback from simulation that the author of a program gets from compiling and early testing.

3 Descriptions of the Case Studies

The first case study was conducted to better understand and maintain a part of the 5ESS that was particularly sensitive to a large flow of internal messages. The behavior of the system was abstracted from the code and from discussions with developers. That is, the design was recovered from the implementation. A model was developed in LOTOS, and analysis of the model revealed some potentially useful modifications of the system.

The second case study was conducted to formalize requirements for call processing behavior. First, a LOTOS model of POTS (Plain Old Telephone Service) was constructed. Next, the Bellcore specifications for call waiting^[3], call forwarding^[2], and conference calling^[4] were used as references in building

LOTOS models of these features. This study was conducted to investigate the applicability of LOTOS to the detection of feature interaction.

The third case study was conducted to derive properties from a high-level design of a new part of the 5ESS. The natural language version of the design was translated into LOTOS, and analysis of the model revealed some potential problems in the design. Also, the dialect of LOTOS used was modified during the project to simplify the tasks of model construction and analysis. Several tools were built to support this dialect.

3.1 Design Recovery Study

3.1.1 Method

In our first project we modeled the behavior of part of the 5ESS as part of a performance study. We used design recovery to build our model, as we were unable to find a suitable design specification. There were two primary sources of information about the design: the code and the engineers who wrote the code.

We were fortunate that most of the code we studied was in one module. Most of the methods and tools that we used were most effective if the scope was narrowed to a few functions. For example, we used a static analysis tool to determine which functions set and used global variables. While the tool provides this information for arbitrarily large collections of functions, we would have found it difficult to examine many definitions and uses of every variable. Instead, most variables were only set and used in a few functions.

We did not use execution traces to study the code. These might have been helpful, particularly if we could have filtered the traces for occurrences of special events. However, we feel that we gained a good understanding of the code from walkthroughs and analysis alone.

Engineers were a particularly rich source of information about the system. Most of the information provided by engineers was in the form of specific scenarios. There were, of course, far too many scenarios to explore individually. However, these scenarios provided a set of tests of the model of code that we developed. Also, the scenario style of presentation reinforced our belief that a simulator would be useful in validating the accuracy of models.

3.1.2 Results

The LOTOS model we constructed was about 900 lines long. We used macros to reduce the size of the text to about 250 lines. The difference is in elaboration of argument lists, each of which was represented by a simple identifier in the shorter version.

We only used four LOTOS operators: prefix, alternation, parallel composition (the general form), and process instantiation. The model was written in a constraint-oriented style. To check the accuracy of the model we used a simulator developed by Colin Fidge^[13]. We also produced graphical representations (trees) of the specification as an aid to under-

standing them. These were particularly useful for discovering missing events.

The main result of our project was the identification of some scenarios that could lead to undesirable behavior. We combined some simple performance data with the scenarios to predict overall performance of the system under those scenarios. For example, knowledge about the frequency and duration of some events allowed us to predict that certain timers would expire. The timers were modeled as simple events, causing the scenarios to complete in predictable patterns.

A negative result was our inability to model some behavior. One communication mechanism that was difficult to model was an event-based scheme. A part of the system could register interest in particular events, so that those (later) events would result in the invocation of specific functions. In principle this is easy to model in LOTOS. Our difficulty was in discovering all the functions that could be invoked by particular events.

3.1.3 Experience

This was our first experience using LOTOS to describe 5ESS. One feature of LOTOS we found awkward to use was process instantiation (similar to a subroutine call). Basic LOTOS provides a renaming mechanism for events that appear in processes. The declaration of a process provides a list of all the events that appear in the process, similar to the argument list of a subroutine. Each instantiation of the process must list all arguments, so that all renamings can be identified. We frequently made mistakes in preparing argument lists, either at process definition or instantiation. One simple trick we employed was to maintain alphabetic order in all argument lists. A better method was to create the lists automatically via macros.

The simulator we used was adequate for small specifications, but bogged down on large ones. Debugging with the simulator was awkward, since little information about the states of processes was available. The tree representations proved to be useful here, as we could manually trace the flow of execution to keep track of each process.

Another feature of LOTOS that we found awkward was the frequent use of tail recursion. Since we were performing design recovery, we often wanted to model the behavior of subroutines present in the code. There is no construct in LOTOS with quite the right semantics for this. An enabled process seems to perform this function, but the "EXIT" expression that would denote return must synchronize with all other processes that have an "EXIT." A subprocess is the best alternative, with the return modeled by an instantiation of the parent process. When several levels of subprocesses are used their arguments (gates) must be known to all processes in the chain. This leads to long argument lists.

3.2 Requirements Study

3.2.1 Method

Our second project was conducted as a requirements specification effort. In particular, we were interested

in the interaction of new requirements with old ones as the requirements evolved. This phenomenon is called the “feature interaction problem” [7].

We started with a description of Plain Old Telephone Service (POTS), which has been formally described several times [12] [18]. Next we added new features as they are described in the Bellcore standards. Some of these features were merely additive: they provided new functionality by means of key presses and signals that were not present in the original system. However, as features were added they began to overlap: new meanings were derived for key presses and signals.

We had hoped that it would be possible to write the new requirements (for the new features) as new constraints on the old LOTOS model (for POTS). This did not work for our model, and we had to rewrite POTS several times as features were added. Instead, we tried writing specifications of different extensions of POTS, and combined these extensions. For example, we developed a description of POTS with call waiting and another description of POTS with call forwarding. Then we composed the two specifications to see how close it came to describing a system with both call waiting and call forwarding.

It is worth elaborating some of the reasons that our original approach did not work. Several changes were made to the original POTS specification to describe the new features that were added:

- The meanings of events changed. For example, an off-hook event in POTS is simpler than an off-hook event when there are waiting calls.
- The number of events increased as new features were added. The new events were usually associated with new phenomena, such as flashing, which did not occur in POTS.
- The number of states of a call increased as we extended POTS. This is a by-product of a richer feature set.

It is likely that any specification method would have trouble accommodating all the changes without some disruption in the original model.

However, some of the changes might have been handled by refinement. For example, events could have been redefined by renaming. This is available in LOTOS, but not in Basic LOTOS. Adding events and adding states was problematical because LOTOS does not allow one to decompose events hierarchically. It would have been useful to have had a mechanism for defining and instantiating a sequence of events. Unfortunately, LOTOS does not have anything that does this for arbitrary instances. The *final* sequence of a process can be replaced by a process instantiation, but a sequence in the *middle* of a process cannot.

Our original plan for combining features assumed that the description of each feature was independent of other features—only the state of the system needed to be identified to describe the resulting behavior. Unfortunately, the only way to identify states in LOTOS

is to define new processes. We could have defined a new process for each new event that occurred, but that would have been unreasonable. The alternative is to guess which sequences of events establish “significant” states, and then hope that the new feature fits those choices. As the number of states grew, this became more and more difficult.

We initially used the same simulator as in the first experiment, but later developed our own simulator. We also used the tree representations as an aid in validating and debugging our specifications.

3.2.2 Results

We discovered several ambiguities in the original Bellcore standards for telephone features. This was especially true for the interaction of features. For example, when a phone is forwarded, does the receiving phone keep all the features it originally had (such as call waiting), does it acquire the features of the phone forwarded to it, or does it get some combination? The LOTOS models were useful for discovering and resolving such ambiguities.

In truth, any formalism would have helped us find ambiguities, but the simulator helped us find many ambiguities (or incompletenesses in the original specification) that we missed in our formalization. The absence of expected choices or the presence of unexpected choices pointed out behaviors that we did not expect. Often these were not specified clearly in the original documents, though reasonable interpretations were easy to find.

We were also able to generate test scripts for use in our 5ESS labs. Scenarios from our specifications were translated into a lab execution language, and the resulting scripts were run. This appears to be a promising technique for testing new features, though the simulations in LOTOS might suffice for most design work.

3.2.3 Experience

As in the first case study, long argument lists were problematical. One reason they grew here was to model different devices. For example, each phone needs its own off-hook event, its own on-hook event, its own power-ringing event, etc. The processes that model the devices only need one instance of each of these events, but the switch process needs to distinguish between all instances.

We had difficulty using the original simulator on all the specifications we developed on this project. Debugging was particularly tough, partly owing to the large namespace of events. It was easy to mistype a name in such a way that a different name was caught.

We were never successful at describing POTS in a way that it could be reused in later specifications. Even after we had developed all the examples, there was no obvious way to factor out POTS from the feature-rich versions.

3.3 Design Study

3.3.1 Method

The last study was a design effort. We translated a high-level design that was written in structured English into LOTOS. This was done in two steps: First we translated each section of the structured English design into LOTOS, and next we merged the sections into one LOTOS specification. Although this may seem like a strange method, it follows the practice used in constructing designs for this system. That is, high-level designs are constructed in separate sections, and the sections are brought together for review.

Although this was a design effort, it did not follow the traditional waterfall model of refinement of requirements to meet implementation constraints. Instead, the previous version of the system imposed most of the real requirements. Thus, the main purpose of the design effort was to show that a predetermined set of processes (most old, but some new) could be developed (either modified or created from scratch) to add new functionality.

3.3.2 Primitive LOTOS

During this project we created our own dialect of LOTOS. We called it Primitive LOTOS, or PLOTOS^[1], since it was more basic than Basic LOTOS. We developed several tools to check our specifications, including an improved simulator.

PLOTOS was designed to simplify the tasks that we had found difficult in our earlier case studies. We knew that our restricted use of the language would allow us to make several simplifications:

- We eliminated argument lists from the syntax, since we could derive them automatically from the rest of the specification.
- We eliminated parallel composition. All the processes (except subprocesses) are assumed to be composed in parallel, and to share all events that are common.
- We eliminated inaction, unobservable actions, successful termination, hiding, renaming, enabling, and disabling, as we did not find a need to use any of these features.
- Events may only be of three kinds in PLOTOS: actions, which are experienced by a single process; messages, which are shared by two processes; or cases, which may be shared by any number of processes. Cases are restricted to occur only as labels of the subclauses of the alternation operator. (A case label is just a shorthand for an event.)

In reducing Basic LOTOS to PLOTOS we were designing a subset that fit our problem domain, the 5ESS software, and our chosen style of specification, constraint-based. We do not claim that our subset

is best in any theoretical sense, merely that it sufficed to reduce the complexity of using the language. Serendipitously, the reduction also made our tool development easier.

The result was a simple language, but it still lacked a desirable property: the ability to detect a large class of simple mistakes. We added declarations of events to the language to check for misspellings and inconsistencies. (By eliminating renaming we could have detected misspellings anyway, but it was more consistent with programming language philosophy to introduce explicit declarations. Since we were trying to make designing look like programming, it made sense to make LOTOS look more like a programming language.) We later realized that declarations allowed us to augment the specifications with other useful information:

- Comments can be added to describe the role of each event in the specification, or to show project-specific information, such as developer responsibility.
- The direction of messages can be declared. Basic LOTOS does not have a way to show that an action is directional. While we do not need this for the simulator, we need it for the message sequence charts that we derive from PLOTOS.
- Additional attributes can be assigned to be used later in displaying various representations.

The event declarations represent an important shared resource of a project. They constitute a data dictionary of the design.

We realized when we created PLOTOS that any deviation from LOTOS was dangerous, particularly if we planned on using anyone else's tools. We were careful to define a language that could be easily translated into a subset of Basic LOTOS. For example, one of our translators produces a dialect that is acceptable to our original simulator.

Appendix B contains the vending machine example in PLOTOS. Appendix C shows the tree representation of the vending machine example. At one time we thought that we would develop a simulator that used this tree representation for its interface. However, we found that message sequence charts were a better description of the phenomena that engineers needed to see while they were using the simulator.

Appendix D shows two message sequence charts for the vending machine example, corresponding to two possible scenarios. Message sequence charts have been proposed as an alternative formalism for specifying protocols^[8]. They are also close to the style of discourse between engineers. That is, we often observed conversations between developers that involved specific scenarios and sequences of events. Our charts have no choice or looping constructs—they are simple paths of execution, similar to a single trace of execution. In general, enough scenarios are generated to guarantee that each path of each process is explored in at least one scenario. We developed a simple language to select specific scenarios.

Appendix E shows a sample session with the simulator. Note that the user is prompted for input only when necessary. (There is a limit to avoid runaway sessions.) The exercise option provides breadth-first exploration of the tree of possible choices. This is convenient when debugging a specification.

We plan to perform other tests of PLOTOS specifications and to develop more tools to aid in refinement toward implementation. For example, we believe that model checking will be useful in detecting deadlocks. A significant advantage of model checking over simulation is the guarantee of coverage of all possible states. This is akin to exhaustive testing of a program. Similarly, checking of temporal logic formulae across all states is like exhaustive testing with embedded correctness assertions.

3.3.3 Results

We discovered several ambiguities and inconsistencies in the natural language version of the design. Most of these flaws resulted from postponement of design decisions. That is, the design was known to be incomplete at the stage when we translated it. Further refinement took place as individual engineers worked on their assigned sections. Unfortunately, this design method requires much communication between engineers as these refinements are made. If decisions are not communicated conflicts will appear during integration and system testing. We feel that an important benefit of using a formal method during design is to help reduce the amount of required inter-developer communication.

As in the call processing study, almost any formalism would have been useful, since it was the process of formalization that detected most errors. In this case the majority of errors were omissions, though some errors were due to different design decisions, such as choices of where to handle specific error cases. Of course, we do not know if we missed any errors that another method might have caught.

We also hoped to assist system testing. While we were able to describe many of the tests, we were unable to implement them automatically due to the problem domain—hardware fault tolerance. Many of the tests involved breaking communication paths, which was usually done with special software and hardware actions.

3.3.4 Experience

We were pleasantly surprised by the improvement in our ability to produce reasonable specifications after we developed PLOTOS and its tools. Declarations of events removed a large class of common mistakes, and they revealed many inconsistencies immediately. For example, we would occasionally choose a name for an event while designing one scenario, and then use a different name for the same event while working on another scenario. This type of inconsistency occurs often in the natural language design, since different people work on different scenarios.

The improved simulator also helped detect more flaws in our specifications. We were able to do much more testing due to the improvements in speed and the simpler user interface. Also, the exercise option allowed more complete exercising of event sequences.

Message sequence charts were particularly helpful in validating the PLOTOS version with the natural language version of the design. Whenever we discussed our work with developers they expressed more interest in message sequence charts than any other representations. Part of this interest comes from previous experience using SDL^[9]. However, message sequence charts also seem closest to the way that 5ESS developers think when designing systems and tests.

4 Some More Lessons Learned

Before conducting the case studies we had several questions about the potential use of LOTOS in development and maintenance:

- Can LOTOS adequately express the important aspects of system behavior for requirements, design and maintenance?
- How difficult is LOTOS to learn and use?
- What tools are needed to support the use of LOTOS?
- How does LOTOS compare with other methods?
- What are the benefits of using LOTOS?

This section will discuss some of our findings in attempting to answer these questions.

4.1 Expressibility of LOTOS

With few exceptions, LOTOS was able to express all the behavior of interest to us. If anything, LOTOS is too expressive, allowing one to say many things that one does not mean. We found some idioms awkward, such as the explicit declaration of event arguments to processes. However, we were able to avoid most of these problems through restriction and simplification of the language.

Although LOTOS is adequate for specification of requirements and designs, it still lacks some features that are important to modern software engineering. First, it lacks modularity constructs. How could libraries of reusable (sub)specifications be declared and used? Second, it lacks redundancy where it would be useful (e.g., event declarations) and requires it where it is most annoying (e.g., procedure arguments).

We should point out that we eliminated a modularity feature of LOTOS when we reduced it to PLOTOS. Specifically, we removed the ability to rename events in instantiations of processes. This facility is similar to the abstraction of formal parameter names for the arguments of subroutines. Unfortunately, we would not have benefited from this abstraction in our examples. The renaming facility in LOTOS is useful for two purposes:

1. when there is more than one instance of a process involved in a protocol, and
2. when there is an abstraction of a process that occurs more than once in a specification.

The first case involves multiple identical instances of a process. We did not need to model this phenomenon in our work. (The ability to generate several instances from a template can be valuable when describing the interaction of several equivalent processes, but all of our processes played unique roles.)

The second case is more subtle, and potentially more common. We tried to use renaming in our requirements study, because we felt that there were several abstractions of call processing that we should be able to factor out. However, we found that the cost of renaming (specifying all the events that occur in a process) outweighed the benefits. Whenever we added or removed an event from a process we needed to change most of the argument lists in the declarations and the instantiations. Of course, judicious use of macros simplifies this type of maintenance, but renaming complicates macro use, also. For example, one must have a macro for every different instantiation of a process, so the benefit of macros starts to evaporate. In the design case study we decided to eliminate renaming, since we were not sure that we would realize any benefits from it.

4.2 Learning and Using LOTOS

We did not find that LOTOS was particularly difficult for developers to learn and use. It does require a paradigm shift from conventional programming languages, but no worse than learning an applicative language, such as LISP. In fact, it is the resemblance to programming that makes designing in LOTOS so attractive. Once we developed the right tools, we found that LOTOS was easy to use.

4.3 Tools for LOTOS

It is as easy to make mistakes writing LOTOS specifications as it is to make mistakes writing programs. One needs the analogue of a compiler to find the obvious syntactic errors, as well as some of the semantic inconsistencies. It is also useful to have a simulator, the analogue of an interpreter for a conventional programming language. However, these tools alone will not guarantee that all errors are discovered.

We found that message sequence charts were especially valuable in validating our specifications. They provide a perspective that closely matches intuition for many applications. Graphical representations of processes were helpful, but we did not find many errors through their use. They were, however, an occasional aid in visualizing the complexity of large processes.

Since we were conducting early studies of feasibility, we could not justify the purchase of industrial-strength LOTOS tools for our investigation. Such tools are comparable to CASE tools in cost, and could be of comparable value as well. We certainly expect to explore their use as our experience grows.

4.4 Comparison with Other Methods

We were often asked by developers how LOTOS compared to other methods of design, such as structured design and object-oriented design. LOTOS is not as mature as most design methods, though that may change as experience grows. One thing that seems to be missing currently is a set of guidelines or heuristics for evaluating LOTOS designs. A common feature of a mature design method is a check at the end of each step to see that the philosophy of the method is being followed.

4.5 Benefits of LOTOS

Clearly, formalism makes some types of analysis possible, where the lack of formalism prevents those analyses. In the case studies we undertook we were able to show several consequences of the specifications we derived. The most important analysis to perform, however, is the simplest. Syntax checking reveals many flaws. Most importantly, it does this immediately, while the author of the document is composing it. Traditional review methods suffer from the postponement of this type of feedback.

Another benefit of LOTOS is its ability to combine constraints easily. We used this technique effectively in the design case study, where composition is the preferred development method. Observation of the consequences of this composition was revealing.

5 Acknowledgments

It is a pleasure to acknowledge the help of my colleagues Don Pham, Susan Eick, and David Furchtgott in the development of the case studies. Colin Fidge donated a copy of his simulator for our early work. Sandra Carrico, Eleftherios Koutsofios, David Ladd, and Peter Mataga developed some of the tools we used.

References

- [1] M.A. Ardis. Primitive LOTOS: A language for describing event synchronization. (in preparation), 1993.
- [2] Bellcore. Call forwarding variable. Technical Report FSD 01-02-1401, Bellcore, 1989.
- [3] Bellcore. Call waiting. Technical Report FSD 01-02-1201, Bellcore, 1989.
- [4] Bellcore. Three-way calling. Technical Report FSD 01-02-1301, Bellcore, 1989.
- [5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
- [6] R. Boumezbeur and L. Logrippo. Specifying telephone systems in LOTOS. *IEEE Communications*, pages 38–45, August 1993.
- [7] E.J. Cameron and H. Velthuisen. Feature interactions in telecommunications systems. *IEEE Communications*, pages 18–23, August 1993.

- [8] CCITT. Recommendation Z.120: Message sequence chart (MSC), 1992.
- [9] J.A. Chaves. Formal methods at AT&T – an industrial usage report. In *FORTE '91 Fourth International Conference on Formal Description Techniques*, pages 85–92, 1991.
- [10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop*, pages 24–37. Springer-Verlag, June 1989.
- [11] J.S. Colson and E.M. Prell. Total quality management for a large software project. *AT&T Technical Journal*, 71(3):48–56, May/June 1992.
- [12] M. Faci, L. Logrippo, and B. Stepien. Formal specification of telephone systems in LOTOS: The constraint-oriented approach. *Computer Networks and ISDN Systems*, 21:53–67, 1991.
- [13] C.J. Fidge. A Basic LOTOS interpreter. Technical Report 140, University of Queensland Department of Computer Science, 1989.
- [14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [15] ISO. *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*. ISO, 1989. International Standard ISO 8807.
- [16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [17] K.J. Turner. A LOTOS-based development strategy. In *Formal Description Techniques II*, pages 117–132, 1990.
- [18] I. Tvrđy. Formal modelling of telematic services using LOTOS. *Microprocessing and Microprogramming*, 25:313–318, 1989.
- [19] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, pages 179–206, 1991.
- [20] W.D. Yu, D.P. Smith, and S.T. Huang. Software productivity measurements. *AT&T Technical Journal*, 69(3):110–120, May/June 1990.

A Basic LOTOS Example

```

process Customer [Chocolate Dime Gum
                  Quarter Think] :=
  Think;
  ( Gum_Customer [Chocolate Dime Gum
                  Quarter Think]
    [] Chocolate_Customer [Chocolate Dime
                           Gum Quarter
                           Think]
  )
endproc

process Gum_Customer [Chocolate Dime Gum
                     Quarter Think] :=
  Dime; Gum; Customer [Chocolate Dime Gum
                       Quarter Think]
endproc

process Chocolate_Customer [Chocolate Dime
                            Gum Quarter
                            Think] :=
  Quarter; Chocolate; Customer [Chocolate
                                 Dime Gum
                                 Quarter
                                 Think]
endproc

process Machine [Chocolate Clunk Dime Gum
                 Quarter] :=
  Dime; Gum; Clunk; Machine [Chocolate Clunk
                              Dime Gum
                              Quarter]

  []
  Quarter; Chocolate; Clunk; Machine
    [Chocolate
     Clunk Dime
     Gum Quarter]
endproc

process root [Chocolate Clunk Dime Gum
              Quarter Think] :=
  Customer [Chocolate Dime Gum Quarter Think]
  | [Chocolate Dime Gum Quarter]
  | Machine [Chocolate Clunk Dime Gum Quarter]
endproc

```

B PLOTOS Example

EVENTS

```

Dime {msg(Gum_Customer, Machine);};
Quarter {msg(Chocolate_Customer, Machine);};
Gum {msg(Machine, Gum_Customer);};
Chocolate {msg(Machine, Chocolate_Customer);};
Think {action(Customer);};
Clunk {action(Machine);};
C_Gum {same: {Customer, Machine};};
C_Chocolate {same: {Customer, Machine};};

```

PROCESS Customer[]

```

Think;
CHOOSE {
  CASE Gum: { Gum_Customer[] }
  CASE Chocolate: { Chocolate_Customer[] }
}

```

SUBPROCESS Gum_Customer[]

```

Dime; Gum; Customer[]

```

SUBPROCESS Chocolate_Customer[]

```

Quarter; Chocolate; Customer[]

```

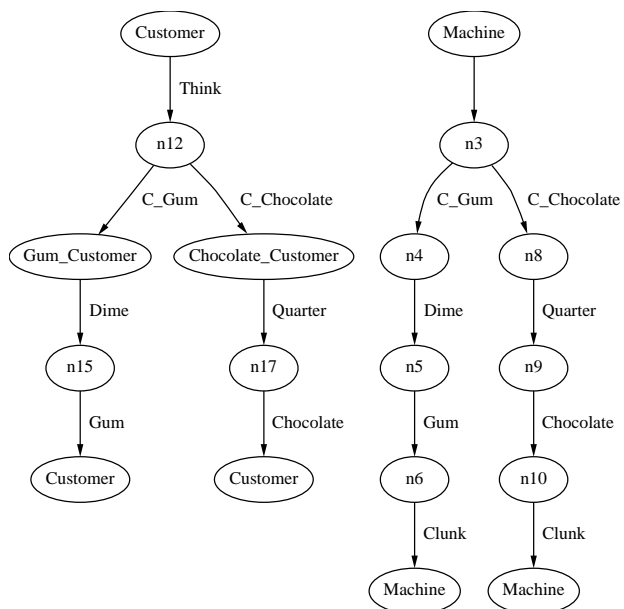
PROCESS Machine[]

```

CHOOSE {
  CASE Gum: { Dime; Gum; Clunk; Machine[] }
  CASE Chocolate: { Quarter; Chocolate;
  Clunk; Machine[] }
}

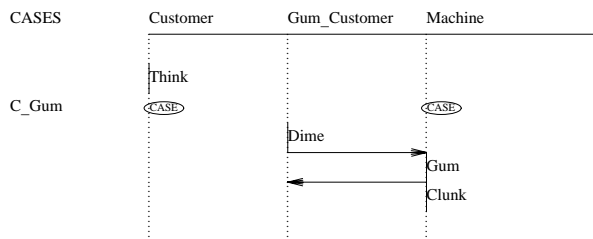
```

C Tree Representation

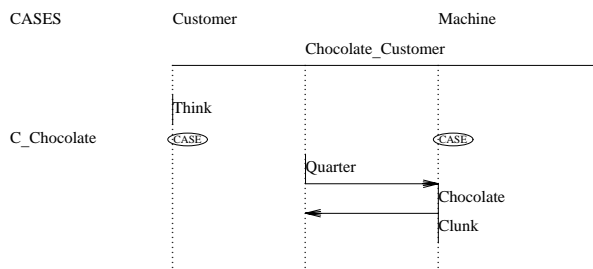


D Example Scenarios

Scenario 1



Scenario 2



E Example Simulation Session

```
Parsing...
  Only one choice...
Think      (Customer) executed
  Possible actions:
    (2) C_Gum      (Customer, Machine)
    (3) C_Chocolate (Customer, Machine)
    (e)xercise, (o)ptions, (q)uit, (r)estart, (s)ave, (v)iew
  Your choice?
C_Gum      (Customer, Machine) executed
  Only one choice...
Dime       (Gum_Customer --> Machine) executed
  Only one choice...
Gum        (Machine --> Gum_Customer) executed
  Possible actions:
    (1) Think      (Customer)
    (8) Clunk       (Machine)
    (e)xercise, (o)ptions, (q)uit, (r)estart, (s)ave, (v)iew
  Your choice? 8
Clunk      (Machine) executed
  C_Gum      (Customer, Machine) blocked
  C_Chocolate (Customer, Machine) blocked
  Only one choice...
Think      (Customer) executed
  Possible actions:
    (2) C_Gum      (Customer, Machine)
    (3) C_Chocolate (Customer, Machine)
    (e)xercise, (o)ptions, (q)uit, (r)estart, (s)ave, (v)iew
  Your choice? q
Finished.
```