On the use of specification styles for automated protocol implementation from LOTOS to C

Peter van Eijk
Harro Kremer
Marten van Sinderen

University of Twente, Fac. Informatics
7500 AE Enschede, NL

June 19, 1991

Abstract

Distributed system design, including implementation and software development, should be based on formal methods in order to achieve correct design. In this paper we study the possibility of automated protocol implementation by transformations from structured formal specifications, in LOTOS, to program code, in C. Particular specification structures are referred to as specification styles. The implementation approach presented here is based on the successful implementation of a medium-scale protocol, the sliding window protocol. Details of this design exercise are included.

1 Introduction

Distributed system design involves architectural design and subsequent implementation phases. So far emphasis in formal approaches for distributed system design has been on the production of 'implementation independent' formal descriptions (FDs) (examples can be found in the OSI arena where FDTs are applied to formally specify services and protocols [vEVD89]) or 'intermediate' implementation FDs. In this paper we address the second half of the design process, which must lead to a final implementation (sometimes called realization): actually running programs. In particular we study the possibilities to automatically derive implementations from FDs. Automated code generation based on formal methods potentially has the advantage of leading to cheaper and higher quality software.

The scope of this study is limited to the use of LOTOS [ISO88, BB87] for FDs and C for final implementations. Even with this limited scope the presented implementation approach is not complete in the sense that it is not applicable to LOTOS FDs of arbitrary protocols. In fact, the approach is based on experience with implementing a number of small toy protocols and one medium-scale realistic protocol, namely the sliding window protocol. How it will scale up for larger and/or more complicated protocols is open to further research.

We believe that the work presented here is new with respect to the 'systematic' use of transformation rules in the implementation approach: it is based on the use of particular FD structures, or specification styles, to facilitate the transformation process. Related work can be found in [MdM89], where protocol implementation from LOTOS to C is explored but without taking advantage of specification styles. A similar approach to eliminating parallelism is related in [CP85].

The structure of the remaining of the paper is as follows. In section 2 an overview is given of a distributed system design methodology that uses specification styles. The relation of this methodology to our work is indicated. Section 3 summarizes the implementation steps of our approach. Two main steps are distinguished, which are discussed in detail in sections 5 and 6. Section 4 introduces the sliding window protocol, which was used as the ultimate test case (so far) for the implementation approach. Details are again presented in sections 5 and 6. Section 5 addresses the transformation from a given protocol FD to a another, more implementation-oriented, FD, and section 6 describes the transformation of the latter to C code. In section 7 the (potential) tool support for the approach is discussed. Section 8 presents an evaluation of the work.

2 Protocol design and specification styles

We will present our approach to protocol implementation in the context of a design methodology that is useful for distributed system design in general. This methodology incorporates the use of a few motivated specification styles that can be used in relationship to each other to support the complete design trajectory from user requirements capturing to software development [VSvS88, VSvSB90].

Requirements capturing serves to determine a system architecture that can be used as the reference for implementations. An architecture should be implementation independent, i.e. it must offer maximum implementation freedom. In practice, however, an architectural specification needs be structured, thus introducing initial implementation choices, in order to keep it comprehensible and manageable. This requires a considered judgement of the architect since choices at this level will affect all implementations.

The architect can be guided by established design principles, such as orthogonality, generality and open- endedness, which in turn must be supported by a suitable specification structuring technique. The *constraint-oriented* specification style is considered to be well suited for this purpose: it enables expression of separate architectural 'constraints' (requirements) as separate specification components which are composed in conjunction.

Subsequent phases of the design trajectory are concerned with the internal organization (the 'how') of the system whose exterior (the 'what') is prescribed by the architecture. As a first (series of) step(s) the construction of the system from interacting subsystems can be established. Besides guidance from the above mentioned design principles, the implementor should use pragmatic and technical criteria to successfully pursue this goal. Here, the *resource-oriented* style proves suitable to structure specifications of (intermediate) implementations. This style enables expression of 'resources' (subsystems) as separate specification components which are composed in conjunction such that internal interactions (i.e. interactions not specified in the architecture) are hidden.

Individual resources can be specified in any style, but iterative application of the above two styles (i.e., re-entering the architectural and implementation phases per resource) is preferable when the resource is of more than elementary complexity. A simple resource can be specified in the *monolithic* style. With this style a collection of alternative sequences of interactions can be specified without any further structure. Alternatively, one can use the *state-oriented* style if investigation of the state space is of concern. Although the state-oriented style is analogous to the monolithic style in the presentation of the dynamic behaviour of a resource, it explicitly shows the states of the resource and how interactions manipulate the state space. Since state-oriented specifications are best suited (compared to other styles) for mapping onto program code, the use of this style is recommended during the final formal step in the design trajectory, before software development is commenced.

In addition to controlling the complexity of distributed system design, specification styles can be used to achieve design correctness. The latter aspect has been illustrated in [vS89] for the 'top' of the design trajectory, using the constraint- and resource-oriented specification styles. Support of tools for transformations in this part of the trajectory are studied in [vE90]. In the following we are concerned with the 'bottom' of the trajectory. Obviously, the resource- and state-oriented specification styles will play an important role in this study.

We will assume a protocol architecture specified in the resource-oriented style with protocol entities identified as separate resources on top of a lower level service. Any specification style can be used for the specification of the protocol entities.

3 Steps in the implementation approach

Our approach starts with selecting a suitable LOTOS specification of the protocol we want to implement. We require that such a specification allows to separate specification components that represent the protocol entities, where each component includes the definition of the user and lower level service interfaces. From the previous section we may conclude that a resource-oriented specification (which could result from earlier design steps) meets this requirement. Alternatively, also a single (not embedded) protocol entity specification, in any style, will do, as long as the interfaces are included.

Next, two main steps can be distinguished:

1. The transformation from the given protocol FD to a more implementation-oriented FD. Because of existing and well-documented experience with implementation from extended finite state machine descriptions, the latter FD can best model a (collection of) state machine(s), with explicit states and state parameters. The specification style of this FD can be considered as a special case of the state-oriented style.

2. The transformation from the state-oriented LOTOS FD to C code. This step involves the change of using an FDT to using a programming language. Given the state-oriented specification, however, it turns out that this is a fairly straightforward step.

Since usually no particular implementation environment is prescribed by the given protocol FD, the implementor is free to choose one. We used a Unix environment to experiment with our approach (notably, the sliding window protocol, which will be introduced in the next section, was implemented on a Unix machine). The Unix interprocess communication facilities (TCP/IP and pipes) are then available to implement a lower level service which can be used by the protocol entities.

At this point, we like to mention three further characteristics of experiments with our approach. First, final implementations did not involve parallelism within protocol entities, i.e. protocol entities were mapped onto separate Unix user processes. In other words, we only implement resources that correspond to full protocol entities. Unix inter process communication thus implements the underlying service, and not arbitrary communication between resources. Secondly, we used the Unix read and write systems calls to implement LOTOS events. And thirdly, abstract data types were implemented manually. More details will be given in the next sections.

4 Test case: the sliding window protocol

We used the sliding window protocol, formally specified by ISO in [ISO89], as a test case to demonstrate and evaluate the above mentioned transformation steps. This selection was based on the following criteria: 1) we wanted to implement a realistic (and not the 'default' alternating bit) protocol, and 2) we required that a complete and 'suitable' LOTOS specification of this protocol was already available.

The sliding window protocol FD describes transmission and retransmission of Protocol Data Units (PDUs) over a single fixed connection (connection establishment and addressing are therefore not included). The (re)transmission procedures of the protocol may use an arbitrary window size to handle both PDU loss and corruption.

Although the FD was complete and structured according to a resource-oriented style, a number of specification errors were revealed by analyzing it with the HIPPO LOTOS simulator [vE89] (these errors were reported back to ISO). For example, a problem was detected with confluency in the data types, and a wrong synchronization was found in the process part related to the acceptance of certain corrupted PDUs.

The corrected FD consists of four parts: data types (271 lines), sender entity (160 lines), receiver entity (73 lines), and medium (31 lines). (The sender entity specification is larger than that of the receiver entity since the first also describes the use of timers for retransmission).

5 Transformation to state-oriented Style

Parallel composition in LOTOS specifications can be used to either express independence of constraints or resources (with the `|||` operator), or to express their dependent concurrent existence (with the `||` operator). In the light of determining the degree of parallelism that the final implementation should support, parallel composition (of resources) requires special consideration of the implementor. It is the freedom inherent in the implementation phase to decide which parallel composed processes in the FD should map onto separate operating system processes. In our case we chose to implement each protocol entity as a separate operating system process (this choice was considered the best from performance point of view, given the sliding window protocol FD and the Unix environment). Other choices, with a higher degree of parallelism, are in principle possible.

The key idea is that in the first implementation step we transform the given FD to an equivalent specification in which no parallelism is expressed in the LOTOS processes that will map onto a separate operating system processes. For this we need a transformation that eliminates parallelism (and therefore also communication between LOTOS processes). To do this we use a technique called *parameterised expansion* [QPF89]. Parameterised expansion can best be described as repeated application of the LOTOS expansion theorem to eliminate all parallel operators in a behaviour expression, but without eliminating value identifiers.

To illustrate the transformation we show the parameterised expansion of a simple duplex buffer (figure 1 and 2).

The result of the expansion is a specification (figure 2) whose structure can be considered as a special case of the state-oriented specification style. Every process definition of this specification consists of a list of alternative events, with each event followed by a process instantiation. Since each process represents a protocol state, this has the effect that all the states of the protocol behaviour are made explicit.

We see that the specification has 4 states, with names `ProDup0` through `ProDup3`. Note also that most states have parameters, hence the name parameterised expansion for the transforma-

```
specification buffer[top, down] : noexit
behaviour
   simplex [top, down] ||| simplex [down, top]
where
  process simplex[gin, gout] : noexit:=
     gin !read ?x : char ; gout !write !x ; simplex [gin, gout]
  endproc
endspec
```

Figure 1: Resource-oriented specification of duplex 1-slot buffer

tion.

It is not always possible to find a finite parameterised expansion of a specification. A necessary precondition is that the specification should be guardedly well-defined. An example that does not have this property is `A = B ||| A`, which recurses on itself without first passing an event. The effect is that an unbounded number of copies of `B` can exist simultaneously. One can argue that such a system cannot be implemented in practice, and would better be specified as `A = B ||| i; A` where the internal step represents an internal decision to allocate resources for another copy of `B`. In a typical protocol example, `B` would be a single connection, and `A` would spawn new connections. This problem does not occur in the given sliding window protocol FD.

Parameterised expansion is also not possible in the presence of process creation loops. An example is `A = B ||| g; A`, where by performing event `g` repeatedly, an arbitrary number of copies of `B` can be created. (Note that the solution to the previous problem can reduce to this.) There are two approaches to this problem. One is to 'ignore' it, and implement the parallelism in another way. Another is to move the problem to the data space, by introducing a parameter to represent the states of all the replicated processes. The given sliding window protocol FD contains process creation loops to model timer behaviour. We used the second approach to resolve this problem: a process was substituted which has a parameter representing the states of all timer processes, and which can therefore 'simulate' all the timers. This works in this particular case, but it is not clear how widely applicable this is.

LOTOS can be used in ways that are hard to implement. A typical example is

```
choice x:SortX [] [z = Operation(x)] -> B(x)
```

where `B` is a behaviour expression parameterised with `x`. Here an arbitrary choice is made for `x` among the solutions of the guard. If there is really only one solution, this can be implemented if an inverse function of `Operation` can be found. Assuming that an inverse function `InvOperation` exists, the above example can then be substituted by `B(InvOperation(z))`. This approach worked for all occurrences of this problem in the given sliding window protocol FD.

The sliding window protocol FD that resulted from this implementation step consists of 3 parallel processes, each of which are in the state-oriented style: sender entity (135 lines), receiver entity (65 lines), and medium (53 lines). The medium process represents the adaption which is needed to bridge the gap between the lower level service in the sliding window protocol and the service that can be obtained by using the interprocess communication facilities of the Unix environment. The adaption reduces the available service in order to simulate PDU loss, etc. (for other protocols or other implementation environments such an adaption may not be necessary). The processes have 8, 6, and 4 states respectively. As follows from the above discussion, the

```
specification buffer[top, down] : noexit
behaviour
   ProDup0[top, down]
where
   process ProDup0[top, down] : noexit  :=
         top  !read ?x_0a:char; ProDup2[top, down] (x_0a)
      []  down !read ?x_0b:char; ProDup3[top, down] (x_0b)
   endproc

   process ProDup1[top, down] (x_11:char, x_12:char): noexit  :=
         down !write !x_11; ProDup3[top, down] (x_12)
      []  top  !write !x_12; ProDup2[top, down] (x_11)
   endproc

   process ProDup2[top, down] (x_21:char): noexit  :=
         down !write !x_21; ProDup0[top, down]
      []  down !read ?x_2b:char; ProDup1[top, down] (x_21, x_2b)
   endproc

   process ProDup3[top, down] (x_31:char): noexit  :=
         top !read ?x_3a:char; ProDup1[top, down] (x_3a, x_31)
      []  top !write !x_31; ProDup0[top, down]
   endproc
endspec
```

Figure 2: Equivalent state-oriented specification of duplex 1-slot buffer

representation of states is not really an intrinsic property of the given specification, but depends on the particular approaches which are chosen to resolve 'implementation problems' and the state matching capabilities of the tools used.

It is not always possible, or desirable, to limit the parallelism as we did for the sliding window protocol. We recall the example of multiple connections, which, in a resource-oriented specification, will be represented as independent (arbitrary interleaved) LOTOS processes. Instead of transforming away this parallelism, it is also possible to expand only a single connection process, and preserve the top level parallelism. In the next section we show how to map top level parallelism to C. As the sliding window protocol does not specify multiple connections, this technique was not used in our approach.

6 Transformation to C

After the first transformation step the resulting FD describes a collection of extended finite state machines. In this section we show how to map this FD to C code. First we describe how a state machine is mapped onto a Unix process. Following that, we discuss issues in the mapping of local interfaces. Subsequently we give some more attention to the way internal events and timers are handled. Finally, abstract data type implementation and multi-process solutions are presented.

In our implementation we chose to map LOTOS events to read or write actions on file descriptors. In Unix both `read` and `write` can block, either waiting for input or waiting for resources. A state machine in a given state waits until it can engage in any event of the *set* of events which are possible in that state. If it can engage in more than one event, an arbitrary choice is made. The selection from a set of events can be implemented in Unix with the `select` system call. This system call takes bit vectors of file descriptors (file descriptor `n` is represented by bit `n`) as argument and returns bit vectors of file descriptors (a subset of those in the argument) on which read or write operations will not block. Hence, the `select` argument can be associated with all the events which are possible in a certain state, in which case the result can be associated with the events which can actually happen. An arbitrary choice from the latter set can then be made. In the case of the sliding window protocol we chose the event that appeared textually first in the FD. The `select` system call itself blocks until either at least one file descriptor can be returned or a certain time period, given as an optional argument, has expired. An interesting property of this solution is that it avoids so-called busy waiting. The corresponding C code is schematically shown in figure 4. A sketch of the LOTOS process from which it is derived is shown in figure 3.

```
process state0[gate] : noexit :=
      event1? .. ; state1[gate]
  []  event2? .. ; state2[gate]
endproc
```

Figure 3: Scheme of a state in LOTOS

```
state0:
  handle internal events;
  determine select-mask;
  select event;
  if (event1) {
    do event1;
    adjust state parameters;
    goto state1;
  }
  if (event2) .....
```

Figure 4: Scheme of a state machine implementation in C

The `select-mask` in figure 4 denotes the argument for the `select` system call. If there are conditions or guards in the FD that govern the possibility of events, these are evaluated in the determination of the `select-mask`. For example, in the sliding window protocol the condition 'window full' prevents the sender entity from engaging in a next service primitive with the local user (thus effecting a back pressure mechanism). This scheme does not work when there are conditions on the *values* that are established in the event.

In LOTOS one can describe very sophisticated interactions. For example, LOTOS allows to specify a synchronous interaction where multiple parties each have their own conditions on the values that can be established in the event. In practical interfaces, events are not necessarily implemented as atomic or synchronous, and putting conditions on their values may not be possible in a given implementation environment. In the sliding window protocol FD, the transmission of a complete (wrapped) PDU is represented by one event. The Unix system calls `read` and `write` are not always capable to support this atomicity. A `read` might result in less data than was asked for. We solve this by buffering the result of a partial `read`. In the state machine this `read` is indicated as failed, and no state change is done. The next time a read is possible the rest of the data is picked up and processed. The interface of a Unix process with the Unix kernel is synchronous, although the kernel buffers data internally. However, as part of the solution to the atomicity problem, we sometimes buffer data ourselves, so our interface with the operating system is not completely synchronous. Why is this correct? In our implementation we have a serial composition of a buffer with the medium (the lower level service). Since the medium also has buffering behaviour, the composition cannot be distinguished from a synchronous interface to the medium.

In a LOTOS specification internal events are used for a number of purposes. The common case, especially in an expanded specification, is that they represent internal communication. According to the LOTOS semantics, they can then be done at any time. Internal events are also used to model expiring timers. In an implementation these appear as an argument in the `select` system call. Another use of internal events is peculiar to service specifications, for instance for modelling PDU loss. In the sliding window protocol case we used a medium process for testing purposes, where these internal events have a certain probability. Finally, internal events are used to model resource constraints. Referring back to an earlier example (on connection creation loops), the condition 'connection table full' would imply that a certain internal event cannot be done. In an implementation these internal events should, if the resource constraint can actually occur, obviously be mapped to the corresponding condition.

In a certain state a number of internal and external events are possible. The LOTOS semantics does not prescribe how to resolve the nondeterministic choice between these. In fact, the LOTOS semantics implies that it does not matter at all, since each choice is a legitimate choice. In our implementation all internal steps, except timers of course, are done first. Following that a `select` is done, with the appropriate time-out. From the set of events associated with the result of the `select` the event is chosen that appears textually first in the specification, as was explained before. It is not essential to do the internal events first. The only implementation error to avoid is to block on a `select` in a state where only internal events are possible.

The implementation of abstract data types was done manually in the sliding window protocol implementation, without any supporting tools. The important data types in the sliding window protocol specification define PDU structures and queues and sets of various objects. For these objects fairly straightforward implementations in C are possible. Each operation in the data types was mapped to a corresponding C function. These functions are 'functional' in the sense that they do not modify their arguments, but make copies of them. It turns out that in this approach the run time of the sender and receiver entity is dominated by copying of data objects.

The sizes of the C code components of the sliding window protocol are: abstract data types 730 lines, sender entity 280 lines, receiver entity 180 lines, medium plus main driver 200 lines, and various interface utilities 290 lines. Three processes are run in the final implementation: the sender, the receiver, and a program to start them up and simulate the medium.

Finally, we like to mention the possibility of multi-process implementations, although this was not used in the sliding window protocol example. If a FD can be written as the arbitrarily interleaved composition of a number of state-oriented specification components (each of which does not contain a parallel operator), there are two straightforward approaches to implementation. One is to map each LOTOS process, or actually each state-oriented component of the specification, on a separate Unix process, and implement its state machine in the way described before. The second approach is to simulate the parallelism. In such a simulation the states of all the processes are collected in a state vector. The `select-mask` is then composed of the select masks of all the processes. From the `select` result a process that can do an event is chosen, and its state is manipulated.

7 Tools

This section discusses the use of tools in the sliding window protocol experiment. First we discuss the usefulness and shortcomings of tools actually used, and then we speculate on what tool functionality may or may not exist to further support the transformation from a LOTOS specification to an implementation.

We used the simulator HIPPO [vE89] to check the correctness of the behaviour of the sliding window protocol specification. As stated earlier this helped to correct a number of errors in the given specification. A limitation of HIPPO is that it is not symbolic, i.e. in every sequence of events values for all variables have to be given. We used the parameterised expansion and state matching of LOLA [QPF89] to transform the sliding window specification to our variant of the state-oriented specification style. The use of LOLA was largely successful, although it has some bugs and shortcomings that necessitated manual intervention. For example, sometimes `B1 || B2` cannot be expanded unless both `B1` and `B2` are both in expanded form. Obviously, LOLA cannot expand process creation loops, so the method for process creation loops given in section 5 had to be applied manually. The output of LOLA is not always syntactically correct or in the required state-oriented style. That problem is usually solved fairly easily. More fundamental improvements are possible in the state matching. State matching is needed to 'fold' an infinite loop of events into a finite description. LOLA does not match commutative variants (e.g. `A[]B` and `B[]A`), so that the expansion generated by LOLA may contain more states than is necessary. A problem of both HIPPO and LOLA is their minimal amount of symbolic ADT computation. For example when `x` is a variable, whose value is unknown, the expression `x and not(a)` is not reduced. Particularly in parameterised expansion, large sets of conditions tend to accumulate, which is sometimes superfluous and can lead to a larger state space.

What other tool functions are conceivable to support the transformation to C code?

- It seems possible to support a substantial part of the mapping from a specification in state-oriented style to the C code that implements the state machine. Nevertheless this tool function would have to make strong assumptions on the mapping of events to Unix and C (e.g. events will be a read or write on a file descriptor), and internal events representing timers and resource constraints would have to be treated separately.

- Once we have a good theory on the elimination of process creation loops by introducing a state vector parameter, tool support for that may be developed.

- There is also room for automatic optimisations. Scope and lifetime analysis of variables can enable the 'overlaying' of state variables. This saves space and assignment statements. Common subexpression detection can reduce repeated evaluation of the same expression.

- Automatic compilation of abstract data types has not been addressed at all here, though approaches to that have been described elsewhere [Heu74].

- It does not seem to be easy to automate the process of reconciling the external interface assumed by the LOTOS specification, and the interface that will actually be used.

8 Evaluation

Our implementation approach from LOTOS to C can never be universal. A tool or method where one can put in a LOTOS specification and get out a C implementation will either need substantial guidance by an implementer or will only work for restricted subset of specifications and very specific target environments. There are two main reasons for this. First, in LOTOS things can be specified that cannot be implemented. For example, using the infinite summation construct, behaviour expressions can express solutions to undecidable problems [vE88]. On the same track, properties of interfaces can be specified that extend beyond the capabilities of the interface that one wants to map to. The typical example is an interaction with complicated conditions on it. Second, the mapping of the abstract interface, specified in LOTOS, to an implementation of it, e.g. system calls, signals, or shared memory, still has to be done, and it is not clear how much of that can be described in LOTOS.

In doing our experiment we have also encountered a number of problems that can be solved at the specification level and at the implementation level. A fairly obvious case that we have mentioned is formed by connection creation loops. The problem can be solved by a LOTOS transformation, but also by a different structure of the implementation. We have mentioned timers, which are usually specified as internal events, and then handled as a separate case in the implementation. Another approach is to make timer control part of the external interface, which would constitute a 'solution by specification' (or more accurately a solution by specifying it away). A final example is getting closer to the external interface. Instead of adapting the target environment to the specification, as was done here, it should also be possible to adapt the specification to the target environment.

9 Conclusions and Future Work

With our sliding window protocol experiment we have shown that it is possible to produce a working implementation of a non trivial LOTOS specification in a systematic way. A tool for parameterised expansion has proved to be indispensable. Though we could do the expansion in this particular case, it is not clear how widely applicable this method is and how it can be extended to cover more complicated specifications. Experiments with other realistic protocol specifications and for different target environments are necessary before we can evaluate the implementation approach properly.

We have indicated in which way tool support can be improved and extended. Better tools will probably allow larger and more complicated specifications to be implemented this way. A fundamental problem that remains is to embed the implementations of the specifications in the target environment. For this we at least need to study the characteristics of existing interfaces. One approach to this is to produce LOTOS specifications of (commercially) available services and interfaces, such as TCP/IP, Berkeley sockets, etc.

References

[BB87] T. Bolognesi and H. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[CP85] Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *ACM SIGGRAPH*, 19(3):199–204, 1985.

[Heu74] T. Heuillard. Compiling conditional rewriting systems. *Lecture Notes in Computer Science*, 308, 1974.

[ISO88] ISO. IS 8807 information processing systems - open systems interconnection - the definition of the specification language LOTOS, 1988.

[ISO89] ISO PDRT 10167 guidelines for the application of Estelle, LOTOS and SDL, June 1989.

[MdM89] J.A. Mañas and T. de Miguel. From LOTOS to C. In K. J. Turner, editor, *Formal Description Techniques - Proceedings of the FORTE 88 Conference*, pages 79–84, Amsterdam, 1989. North-Holland.

[QPF89] J. Quemada, S. Pavon, and A. Fernandez. Transforming LOTOS specifications with LOLA - the parameterised expansion. In K. J. Turner, editor, *Formal Description Techniques - Proceedings of the FORTE 88 Conference*, pages 45–54, Amsterdam, 1989. North-Holland.

[vE88] Peter van Eijk. *Software Tools for the Specification Language LOTOS*. PhD thesis, Twente University of Technology, Enschede Netherlands, 1988.

[vE89] Peter van Eijk. The design of a simulator tool. In P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 351–390. North-Holland, Amsterdam, 1989.

[vE90] Peter van Eijk. Tools for LOTOS specification style transformation. In S. T. Vuong, editor, *Formal Description Techniques, II - Proceedings of the FORTE 89 Conference*, pages 43–52, Amsterdam, 1990. North-Holland.

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS - results of the ESPRIT/SEDOS project*. North-Holland, Amsterdam, 1989.

[vS89] M. van Sinderen. A verification exercise related to specification styles in LOTOS. Technical Report INF-89-18, University of Twente, Enschede Netherlands, 1989.

[VSvS88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In K. Sabnani and S. Aggarwal, editors, *Proceedings of the eighth international conference on protocol specification, testing and verification*, pages 189–204, Amsterdam, 1988. North-Holland.

[VSvSB90] C. A. Vissers, G. Scollo, M. van Sinderen, and H. Brinksma. Specification styles in distributed systems design and verification. *to appear in Theoret. Comput. Sci. special issue dedicated to Tapsoft 89*, 1990.