A Balancing Act: Analyzing a Distributed Lift System

Jan Friso Groote
Eindhoven University of Technology
Technical Applications, Computing Science Department
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail: `jfg@win.tue.nl`
Phone: +31 40 247 5003
Fax: +31 40 2468508

Jun Pang & Arno Wouters
Centre for Mathematics and Computer Science (CWI)
Department of Software Engineering
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: `Jun.Pang@cwi.nl, Arno.Wouters@cwi.nl`
Phone: +31 20 592 4221
Fax: +31 20 592 4199

Abstract

The process-algebraic language μCRL is used to analyze an existing distributed system for lifting trucks. Four errors were found in the original design. We propose solutions for these problems and show by means of model-checking that the modified system meets the requirements.

1 Introduction

As is well known, distributed systems form a major aspect of system design. Verifying the correctness of the protocols that regulate the behavior of such systems is usually a formidable task, as even simple behaviors become wildly complicated when they are carried out in parallel.

Algebraic approaches to the study of concurrent systems focus on the manipulation of process descriptions. Processes are represented by means of process terms consisting of process names, action terms (which represent atomic activities) and operators (specifying the order in which the activities can be carried out). A set of axioms specifies how process terms can be manipulated in such way that the processes they represent are in a certain sense the same.

Traditional process algebras such as CCS [Mil89], CSP [Ros98] and ACP [Fok00] are well suited for the study of elementary behavioral properties of distributed systems. However, when it comes to the study of more realistic systems, these languages turn out to lack the ability to handle data adequately.

In order to solve this problem the language μCRL [GP95] has been developed. This language combines the process algebra ACP with equational *abstract data types* [LEW96]. This is done by parameterizing action and process terms with data. A conditional (if-then-else construct) can be used to have data influence the course of a process, and alternative quantification is added to sum over possibly infinitely many data elements of some data type. Also communication and recursion can be data-parametric in μCRL.

To each μCRL specification there belongs a *process graph*, being a directed graph in which the states are process terms and the edges are labeled with actions. If this process graph consists of finitely many states, then the μCRL tool set [Wou00] can be used in combination with the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) [FGK+97] to generate, visualize and analyze the process graph. For example, one can detect the presence of deadlocks and livelocks, single step through the graph, and apply model checking [CGP00] to check the validity of temporal logic formulas.

This paper reports on the analysis of a real-life system for lifting trucks (lorries, railway carriages, busses and other vehicles). The system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own microcontroller. The controls of the different lifts are connected by means of a circular network. A special purpose protocol has been developed to let the lifts operate synchronously.

This system has been designed and implemented by a Dutch company. When testing the implementation the developers found three problems. They solved these problems in an *ad hoc* manner, although the causes of two of the three problems were unclear. Moreover, the developers were unsure that there were no other bugs hidden in the system. In close cooperation with the developers, we specified the lift system in μCRL; we strove to stay as close as possible to the actual implementation. Next, we analyzed the resulting specification with the μCRL tool set and CADP. The three known problems turned up in our specification (which indicates that our specification is valid). In addition we found a fourth error. This error was unknown and found its way into the implementation of the lift system. We propose solutions for these problems. We have analyzed the μCRL specification that results from the incorporation of the proposed solutions, showing that this specification meets the requirements of the developers.

This article is set up as follows. After this introduction, we give an informal specification of the lift system (Section 2). Next we discuss the requirements which the system should satisfy (Section 3) and the methods we used to verify these requirements (Section 4). Then, we report on the problems we found and we propose solutions to the problems (Section 5). We conclude by drawing some conclusions (Section 6). Our initial specification, the modified specification, the transition systems generated with the μCRL tool set and the formulas used for model checking can be found on the world wide web at `<http://www.cwi.nl/~mcrl/lift/>`. More details and explanations of the specification and analysis can be found in [GPW01].

2 Description of the lift system

First, we explain the general layout of the lift system (Section 2.1). Then we explain the manner in which lift movement is controlled (Section 2.2).

2.1 Layout of the lift system

The system studied in this paper consists of an arbitrary number of lifts. Each lift supports one wheel of a vehicle being lifted. The system is operated by means of buttons on the lifts. There are four such buttons on each lift: UP, DOWN, SETREF and AXIS. The system knows three kinds of movements. If the UP or DOWN button of a certain lift is pressed, all the lifts of the system should go up, respectively down. If the UP or DOWN button is pressed together with SETREF, only one lift (the one of which the buttons are pressed) should go up or down. This allows the operator to adjust the height of a lift to inequalities in the surface of the floor. If the UP or DOWN button is pressed together with the AXIS button, the opposite lifts (and only those) are supposed to move up or down, respectively. This is needed to replace the axis of a truck. As different trucks may have different numbers of wheels, the operator may add or remove lifts to or from the system. We have only studied the first kind of movement and for that reason we will restrict the remainder of the paper to that mechanism.

Normally, the lifts contain a locking pin which is intended to prevent the lift from moving down when motors fail, or oil is leaking from the hydraulic pumps or valves. This pins restrict the movement of the lifts. If one wants to move the lifts over a larger distance this pin has to be retracted. This detail is not taken into account in our specification.

Lift movement is controlled by means of a microcontroller. The lift controller can adopt eight different states. For our study the following states are important: STARTUP, STANDBY, UP, and DOWN. The meaning of these states will become clear in the course of the discussion.

The lift controls of the different lifts belonging to a system are connected to a 'circular' CAN bus [Rob91] which is interrupted by relays (see Figure 1). The different controllers connected to the bus are called 'stations'. There is a relay between every pair of adjacent stations and each relay is controlled by the station at the left side. When the system is switched on all relays are open. After initialization, all relays but one should be closed. In effect this means that if the initialization succeeds all the stations are connected to one linear bus.

The CAN bus is a simple, low-cost, multi-master serial bus with excellent error detection capabilities. Multi-master means that all stations can claim the bus at each bus cycle and several stations can claim the bus simultaneously, in which case a non-destructive arbitration mechanism determines which message is transmitted by the bus. A message on the bus is immediately received by all other stations connected to the sending station via closed relays. The CAN protocol does not use addresses.

In the lift system, the user data field of the messages transfered over the bus contain three pieces of information: the position of the sender station, the type of the message, and the (measured) height of the sender's lift. There are two kinds of messages: SYNC messages and 'state' messages. State messages report the state of the sender station (e.g., STARTUP, STANDBY, UP, DOWN). SYNC messages initiate physical movement. In response to a SYNC message each

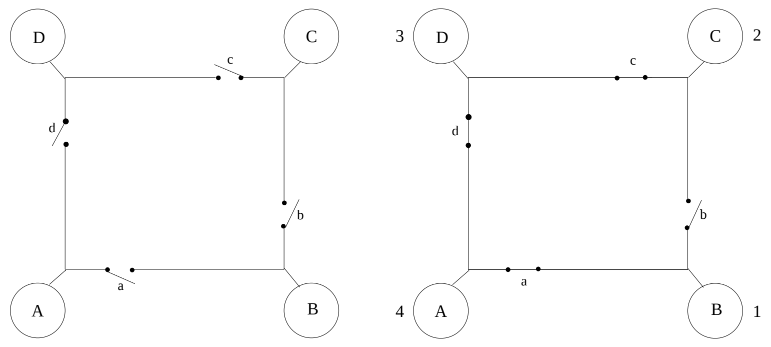

Figure 1: State of the relays before (left) and after (right) initialization

station will immediately transfer its state to the input of the motor of its lift. This means that if the station is in the UP state after a SYNC message, the lift will move up a fixed distance; if the station is in the DOWN state, the lift will move down a fixed distance; and if the station is in STANDBY it will not move.

The system continuously checks the heights broadcasted in the messages to determine if they do not differ too much. If there is something wrong an emergency stop is brought about. This is not modeled in our specification.

2.2 Control of lift movement

To assure that all lifts move simultaneously in the same direction, the station initiating a certain movement must verify whether all stations are in the appropriate state before it sends the SYNC message.

The CAN protocol allows several stations to claim the bus at the same time. However, in the lift system, the controls are programmed in such a way that (during normal operation) the stations take turns claiming the bus. They claim the bus in a fixed order (turn left in Figure 1).

To achieve this orderly usage of the bus, each station must know its position in the network. Furthermore, in order to be able to find out whether all stations are in the same state, each station must know how many stations there are in the network. This is achieved by means of a startup phase in which all the stations come to know their position in the network as well as the total number of stations in the network. This startup phase is discussed in Section 2.2.1, normal operation in Section 2.2.2.

2.2.1 Startup

As said, when the system is switched on, all the relays are open (see the left part of Figure 1).

In the startup phase two things might happen to a station:

- The SETREF button of that station might be pressed. In this case the station will initiate the startup phase as follows:

    1. it stores that it has position 1,
    2. it adopts the STARTUP state,

3. it closes its relay,
4. it broadcasts a STARTUP message,
5. it opens its relay,
6. it waits for a STARTUP message,
7. it stores the position of the sender of that message as the number of stations in the network,
8. it adopts the STANDBY state,
9. it broadcasts this state.

- The station might receive a STARTUP message from another station. In this case:
  1. it adds 1 to the position of the sender of that message and stores this as its own position,
  2. it stores its own position as the number of stations in the network,
  3. it adopts the STARTUP state,
  4. it closes its relay,
  5. it sends a STARTUP message,
  6.
     - if it receives a STARTUP message it stores the position of the sender of that message as the number of stations in the network,
     - if it receives a STANDBY message it adopts the STANDBY state (if the station has position 2 it will in addition initiate normal operation by broadcasting its state).

Assume, for example that in the system of Figure 1 the SETREF button of lift B is pressed. The station of this lift gets position ('logical address') 1. It closes the relay between B and C, broadcast a STARTUP message, and open this relay again. The STARTUP message from B is received by only one station (C). This station draws the conclusion that it has position 2. It subsequently closes the relay to D and broadcasts a STARTUP message. This message is received by only one station (D). This station draws the conclusion that it has position 3, closes the relay to A and sends a STARTUP message. This message is received by A and C. C draws the conclusion that now there are three stations in the network. A draws the conclusion that it has position 4, closes the relay to B and broadcasts a STARTUP message. This message is received by B, C, and D. C and D draw the conclusion that now there are four stations in the network. Station B draws the conclusion that the circle is completed. It stores the position of the sender of that message (4) as the number of stations in the network, adopts the STANDBY state and initiates normal operation by sending a STANDBY message. This message is received by C, D, and A which adopt the STANDBY state in response.

The result is that all stations are connected in the manner pictured in the right part of Figure 1, that all stations know how many stations there are in the network and what their position is, and that all stations are in STANDBY. Normal operation starts when station 2 broadcasts its state.

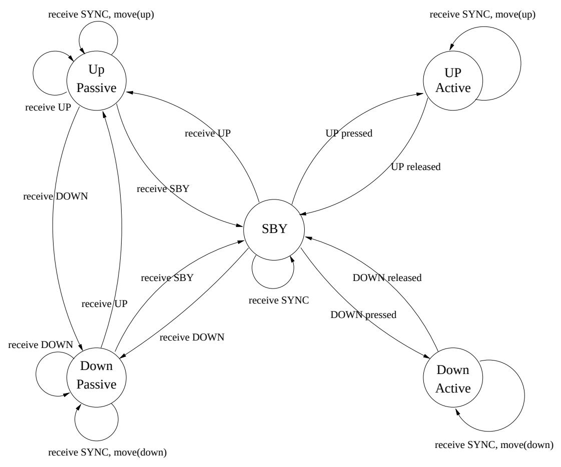

Figure 2: State transitions of an individual lift during normal operation

2.2.2 Normal operation

During normal operation, the first station broadcasts its state and height, then the next station broadcasts its state and height and so on, until the last station has broadcast its state and height after which the first station starts again.

The transition diagram of each lift during normal operation is sketched in Figure 2. Initially all stations are in STANDBY. A station in STANDBY changes to another state if one of its buttons is pressed or if it receives a message with another state. The station that is initiating a certain change (i.e., when it is in STANDBY and a button is pressed) is called the active station. All other stations are passive. If the UP or DOWN button of a certain lift is pressed and its station is in STANDBY that station becomes active and changes its state to UP or DOWN, respectively. When a passive station receives a state message it adopts the state in that message. An active station does not change its state in response to state messages. The state of an active station changes only if the appropriate button is released. In that case its state changes to STANDBY and the station becomes passive again.

As said, physical movement is initiated by a SYNC message. In order to assure that all lifts move in the same direction the active station will count the number of messages that contain the intended state. The active station will send a SYNC message if and only if it has counted enough messages with the right state when it is its turn to use the bus.

Assume, for example, that all stations are in STANDBY and that the UP button of station 4 is pressed. This station adopts the UP state. When it is this station's turn to use the bus it will broadcast its state; in response the other stations will adopt the UP state too. Next, it is station 1's turn to use the bus. This station will broadcast its state (which is UP). The message from station 1 is received by all other stations, among which the active station 4. As the state in the message is the same as that of the active station 4, this latter station will count this message. In the next two cycles station 2 and station 3 claim the bus in turn and broadcast their states (UP), both messages are counted by station 4. So, if all goes well, station 4 will have received the right number of UP messages when it is its turn to use the bus again and it will send a SYNC message to initiate physical movement.

3 Requirements

There are five requirements for the lift system. Each requirement describes a different aspect of the system's behavior.

1. *Deadlock freeness*: the lift system never ends up in a state where it cannot perform any action.
2. *Liveness I*: though the system might ignore buttons temporarily, it is always possible to return to a state in which pressing the UP or DOWN button of any lift will yield the appropriate response.
3. *Liveness II*: if exactly one UP or exactly one DOWN button is pressed and not released, then all the lifts will (eventually) move up or down, respectively.
4. *Safety I*: if one of the lifts moves, all the other lifts should simultaneously move in the same direction.
5. *Safety II*: if the lifts move, an appropriate button is pressed. In other words, the lifts will not move if no one has pressed an appropriate button.

4 Methods

We specified the lift system in μCRL. As is demonstrated by this case study, this language is useful as a tool to analyze medium-sized distributed systems.

The language μCRL has been the basis for the development of a proof theory (the *cones and foci method*) [GP94] that has enabled the formal verification of distributed systems in a precise and logical way. When proving a system correct the axioms of μCRL are applied to a μCRL specification of that system to show that that specification is identical to a specification of the intended external behavior. Proving the correctness of a system by hand is an elaborate and time consuming task which is infeasible for large systems.

In our case, proving is not an option, not only because of the size of the specification but also because the intended external behavior is difficult to describe. For that reason we studied the system by model checking.

Model checking is an automatic technique to determine whether a state transition system satisfies certain requirements [CW96]. It has been successfully applied to a large number of communication protocols, such as the link layer protocol of the futurebus+cache coherence protocol [CGH$^+$93], the IEEE 802.3 Ethernet CSMA/CD protocol [NS94] and the ACCESS.bus protocol [BG96]. In order to check whether a certain requirement holds, it should first be expressed as a temporal logic formula. A model checker searches the reachable states of a certain labeled transition system to determine whether this formula holds. If the model checker finds that the formula does not hold it presents a fragment of the state space that violates the requirement.

In our study, the μCRL tool set was used to generate a transition system from the initial μCRL specification. This transition system was analyzed with the CADP tool set. When an error was found the specification was modified and the modified specification was analyzed again.

A notorious problem when model checking is the *state explosion* caused by the fact that the number of states grows exponentially with the number of components of a distributed system. One way to fight the explosion of states, is to abstract away from the internal behavior of a system. In line with this approach we rename all internal behavior into the silent action τ, and consider the resulting process graph modulo weak equivalence. This allows an efficient minimization of the state space [Mil89].

5 Results

Four errors were found in the original design. We discuss these problem separately and propose solutions (Sections 5.1–5.4). The modified specification resulting from the incorporation of our suggestions was shown to meet the requirements (Section 5.5).

5.1 Problem 1

The first problem occurs if station 2 sends a STARTUP message before the relay between station 1 and 2 is opened. This STARTUP message is received by station 1 which will draw the erroneous conclusion that the circle is completed. From this all sorts of errors may occur (depending on the exact timing). For example, if the relay between station 1 and station 2 is closed before station 1 sends the SBY message which initiates normal operation no station will receive this message. The start up phase will continue as intended until station 1 receives the STARTUP message from the last station in the system. As this is unexpected it will adopt the STOP state, resulting in a deadlock.

The developers had spotted this problem in the test phase but they were unaware of its cause. They had solved the problem by adding delays[1] before sending a STARTUP message. Our experiments indicate that this solves the problem adequately (if the delay is long enough to make sure that the relay between 1 and 2 is opened before station 2 sends the STARTUP message). They also indicate that it suffices to delay only the second STARTUP message.

5.2 Problem 2

The second problem occurs if the SETREF buttons of two lifts are pressed at the same time. This may result in different lifts moving in different directions. Assume that the system consists of four lifts (A, B, C, D) and that the SETREF buttons of A and C are pressed at the same time. Both A and C send a STARTUP message which is received by respectively B and D. The relays between A and B, and between C and D are opened again. Next B closes the relay between B and C and then B broadcasts a STARTUP message. This message is received by C. Station C draws the conclusion that the circle is completed and initiates normal operation. At the same time D closes the relay between D and A and

[1]In μCRL specification, the delay is modelled by the communication of two actions.

sends a STARTUP message that is received by A, after which A initiates normal operation. The result is that there are two independently operating networks, one consisting of A and D; the other of B and C. There is no way in which the stations or the bus can prevent or detect this situation.

A similar situation may occur if the SETREF buttons of two adjacent lifts (say A and B) are pressed. Assume that B sends a STARTUP message slightly before A does so. The message from B is received by C. Assume that next the relay between B and C is opened again and that A subsequently sends its startup message. Station B receives it and draws the conclusion that the circle is completed and initiates normal operation. Station A opens the relay between A and B, and after receiving a STARTUP message from D it finishes the startup phase. The result is that B is isolated from the rest of the network. Again the system will not detect this error.

Given the chosen bus it seems impossible to solve this problem satisfactorily. The developers choose to emphasize in the manual that it is important to make sure that in the initial phase the SETREF button of only one lift is pressed.

We have modified the specification in such way that it is impossible to initiate the system by pressing the SETREF button of several lifts at once. In our opinion this is a dangerous assumption, but the developers have a different opinion.

5.3 Problem 3

The third problem occurs if a button is pressed and released at the wrong moment. Suppose that in a network of four stations all stations are STANDBY, and that the DOWN button of station 1 is pressed as a result of which it acquires the DOWN state. When it is the turn of station 1 to use the bus it broadcasts the DOWN state, and all other stations adopt this state in response. Suppose that the DOWN button is released after station 3 sends its DOWN message, but before station 4 has done this. As a result station 1 returns to the STANDBY state. In this state it adopts the state of all state messages it receives, so when station 4 sends its state message it adopts the DOWN state. We now have the situation that all stations are in DOWN state, but there is no active station. This means that they will remain in that state until the system is shut down.

This problem was discovered by the developers when testing the system and they solved it by means of an initiator flag. In our example this flag is set in station 1 if the DOWN button is pressed. When the flag is set, the station does not accept state changes from the bus until both its own state and the received state are STANDBY. Then the flag is reset and new commands are accepted. A simpler solution would be to let the station wait to become passive after the button is released, until it is that station's turn to use the bus. This is the solution incorporated in our modified specification.

5.4 Problem 4

The fourth problem occurs when two buttons on different lifts are pressed at the same time. Suppose there are four stations in the network and that the DOWN buttons of station 1 and station 2 are pressed at the same moment as the result of which both stations become active. Assume that it is station 1's turn to use the bus. It sends a DOWN message, and in response station 3 and station 4 adopt the DOWN state. In turn stations 2, 3 and 4 send a DOWN

Number of lifts	states generated	transitions generated	states reduced	transitions reduced	CPU time generation
2	391	742	67	162	5s24
3	7,369	19,245	346	1110	18s91
4	129,849	422,884	1,317	5,300	5m15s95
5	2,165,446	8,723,465	4,256	20,680	3h07m00s00

Table 1: Transition system dimensions

message. When it is the turn of station 1 to use the bus again, it has counted three DOWN messages so it sends SYNC (after which all lifts move down), and as the DOWN button is still pressed it then sends DOWN. Now it is station 2's turn and as this station is active and has counted three DOWN messages it sends a SYNC message. Suppose (and now comes the problem) that the DOWN button of station 1 is released after station 1 has sent the DOWN message and before station 2 sends the SYNC message. As a result station 1 is in STANDBY when it receives the SYNC message, and its lift remains at the same height while the others move down.

A similar problem occurs if the UP button of station 2 is released just after station 3 has sent its DOWN message but before station 1 sends its SYNC message. In this case lift 2 will remain at the same height while the others move down.

This problem was not known to the developers and found its way into the implementation. We propose to solve this problem by allowing a station to become active only when it is its turn to use the bus and only when at that moment there is no other station active (i.e., when the message from the previous station is STANDBY).

5.5 Verification of the modified specification

All five requirements stated in Section 3 were shown to be satisfied by modified specifications of systems with respectively 2, 3, 4 and 5 lifts.

The dimensions of the generated state spaces are summarized in table 1. For each of the lift systems, the size of the generated transition system, the size of that system after reduction modulo weak bisimulation and the time it took to generate the system are given. Generation was performed on a 300 MHz SUN Ultra 10 Processor with 1024Mb memory.

6 Conclusion

Let us take stock. We have discovered four real errors in the design of a real system. The fourth was unknown and has found its way into the final release. Three of these problems were also found by the developers of that system in the test phase. For two of the three known problems, it was only known that the problem occurred but not what its causes were. We have discovered the causes of those problems. We were able to solve three of the four problems and showed by means of model checking that the modified system meets the requirements. The problem 2 is difficult to solve within the restrictions of the chosen hardware.

7 Acknowledgements

We like to thank Wan Fokkink for comments on earlier versions of this paper and Izak van Langevelde for help in model checking as well as comments on earlier versions.

References

[BG96] Bernard Boigelot and Patrice Godefroid. Model checking in practice: an analysis of the access.bus protocol using spin. In *Formal Methods Europe'96, Oxford*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478. Springer-Verlag, March 1996.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. In L.Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2000.

[CW96] E.M. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[FGK+97] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings 8th Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1997.

[Fok00] W. J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. Springer-Verlag, 2000.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing Series, pages 231–250. Springer-Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[GPW01] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. Technical Report SEN-R0111, CWI, Amsterdam, 2001.

[LEW96] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[NS94] V.G. Naik and A.P. Sistla. Modeling and verification of a real life protocol using symbolic model checking. In D.L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 194–206. Springer-Verlag, 1994.

[Rob91] Robert Bosch Gmbh, Postfach 30 02 40, D-70442 Stuttgart, Germany. *CAN Specification. Version 2.0*, 1991.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Wou00] A. G. Wouters. *Manual for the μCRL Toolset.* Department of Software Engineering, CWI, 2000.