Analysis of a Distributed System for Lifting Trucks

J.F. Groote
email: jfg@win.tue.nl
Eindhoven University of Technology
Section Technical Applications, Computing Science Department
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

J. Pang & A.G. Wouters
email: Jun.Pang@cwi.nl, Arno.Wouters@cwi.nl
Center for Mathematics and Computer Science
Cluster of Software Engineering
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
The process algebraic language μCRL is used to analyse an existing distributed system for lifting trucks. Four errors are 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.

2000 Mathematics Subject Classification: 68N30 [Specification and verification]; 68Q60 [Model checking]
1998 ACM Computing Classification System: D.2.1 [Requirements/Specifications]; D.2.4 [Model checking]; C.2.2 [Protocol verification]
Keywords and Phrases: μCRL, model checking, process algebra, protocol verification, specification
Note: This research is supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

1. Introduction

As is well known, protocols for 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.

Process algebras such as CCS [Mil89], CSP [Ros98] and ACP [BW90, 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 parameterising 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 *transition system*, in which the states are process terms and the edges are labelled with actions. If this transition system consists of finitely many states, then the μCRL tool set [Wou01] can be used in combination with the CÆSAR ALDÉBARAN DEVELOPMENT PACKAGE (CADP) [FGK+97] to generate, visualise and analyse this transition system. For example, one can detect the presence of deadlocks and livelocks, step through the graph, and apply model checking [CGP00] to check the validity of temporal logic formulae.

This paper reports on the analysis of a real-life system for lifting trucks (lorries, railway carriages, buses 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 cyclical 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 by trial and error, partly because the causes of two of the three problems were unclear. In close cooperation with the developers, we specified the lift system in μCRL; we generally strove to stay as close as possible to the actual implementation. Next, we analysed the resulting specification with the μCRL tool set and CADP. The three known problems turned up in our specification (which adds to our confidence that the specification is close to the actual implementation). In addition we found a fourth error. This error was unknown and found its way into the implementation of the lift system. We incorporated solutions for these problems in the specification. We have analysed 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 structured as follows. After this introduction, we give an informal description of the lift system (Section 2). Next we discuss the requirements which the system should satisfy (Section 3) and the initial specification in μCRL (Section 4). Then, we report on the problems we found and we incorporated solutions for these problems in the specification (Section 5). The methods and result of our formal verification is presented in (Section 6). We draw some conclusions in (Section 7). A short version of this paper has been presented in *the 6th International Workshop on Formal Methods for Industrial Critical Systems* (FMICS 2001) [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.

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 'cyclical' CAN (Controller Area Network) 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 its left side.

The CAN bus is a simple, low-cost, multi-master serial bus with 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 transferred 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 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 as this would increase the number of states of the system tremendously. If there are n possible values for the heights (typically $n = 256$), and m lifts, an increase of at least a factor n^m can be expected.

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 stations 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 counter clockwise 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 below:

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,

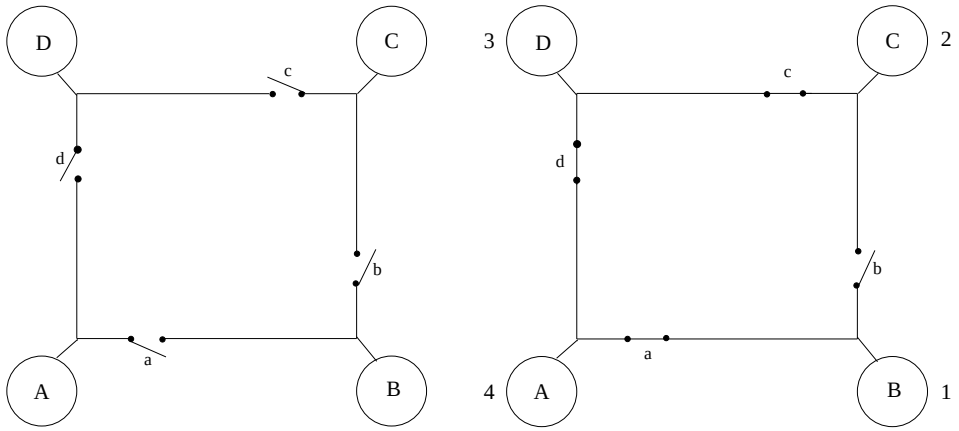

Figure 1: State of the relays before (left) and after (right) initialization

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 an another 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 station B is pressed. The station of this lift gets position 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.

Normal operation During normal operation, the first station (with position 1) 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.

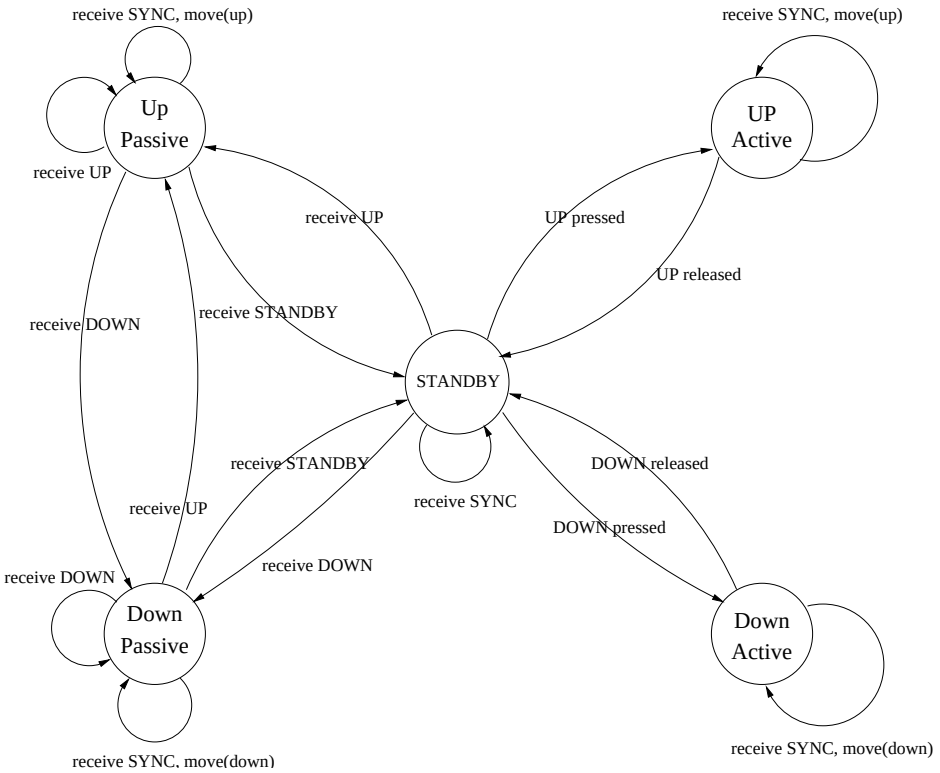

Figure 2: State transitions of an individual lift during normal operation

The transition diagram of each lift during normal operation is sketched in Figure 2 [1]. 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 pressed 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 (i.e., all the other stations are in the same state as itself), 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 (getting a message from its predecessor), it will broadcast its state; in response the other stations will adopt the

[1] Some actions of pressing or releasing a button are not represented in this figure, since those actions do not make any state transition of a lift during normal operation phase.

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, 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 that have been formulated in cooperation with the developers. 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*: it is always possible for the system to get 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. Specification

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

4.1 A short introduction to μCRL

The specification language μCRL is based on the *process algebra* ACP extended with a formal treatment of data. A μCRL specification consists of two parts. One part specifies the data types, second part specifies the processes. Each data type is declared using the keyword **sort**. Elements of a data type are declared by using the keywords **func** and **map**. The keyword **func** is used to declare the functions that construct a certain sort. The additional properties or additional relations between the elements of an already defined sort are declared with the keyword **map**. They are defined by means of equations, which consist of an optional variable declaration (starting with the keyword **var**) followed by an equation section (starting with the keyword **rew**). For example, the sort `Bool` of booleans with conjunction and negation is defined as follows:

```
sort  Bool
func  T,F: →Bool
map   and: Bool×Bool→Bool
      not:Bool→Bool
var   b: Bool
rew   and(T,b)=b
      and(F,b)=F
      not(T)=F
      not(F)=T
```

Because booleans are used in the 'if-then-else' construct of the process descriptions, the sort `Bool` must be included in every μCRL specification. Besides the declaration of the sort `Bool`, it is also obligatory that `T` and `F` are declared in every specification. To reflect equality between terms, one

needs to specify an equality function eq:D× D→ Bool. Actually, such an equality function is only needed for data types that are used as parameters of actions that occur in a communication. The equality relation is achieved by comparing every two elements. (See an example in the definition of the sort Address, and note this is only feasible for finite date types.) For other data types in our discussion, the specification of equality function eq is omitted to increase the readability and save space.

The definition of a process is constructed from action names, process names and process algebraic operators. Actions are declared by means of the keyword **act**. They are defined with zero or more parameters. Each of the parameters is defined over some data type. There are two predefined actions in μCRL: δ represents deadlock, τ represents the hidden actions[2]. Processes are represented by process terms. Process terms describe the order in which the actions may happen. Process terms consist of basic process terms (action names and process names) combined by process algebraic operators. p·q indicates sequential composition, where p and q are processes. The operator '+' stands for non-deterministic choice. p+q means that it can behave as p or q. The parallel composition of p and q is written as p||q. The **sum** operator provides the possibly infinite choice of an element of a sort. The conditional expression 'if-then-else' has the form of p◁b▷q[3], where b is a data term of sort Bool. This means that if b holds then it behaves as p, otherwise as q. The keyword **comm** can be used to specify which actions may synchronise. Two actions can only synchronise if their data parameters are semantically the same, which means that communication can be used to represent data transfer from one process to another. If two actions are able to synchronise, then we only want these actions to occur in communication with each other, and not on their own. This can be done by the operator **encap**. The operator **hide** hides all enclosed actions by converting them into the τ action. It is an important means to analyse communicating systems. It makes the internal behaviors invisible. The initial behavior of the system can be specified with the keyword **init**. The syntax and semantics of μCRL are given in [GP95].

As we described in Section 2, our specification is still an abstraction of the real system. Such details as the locking pins, the parameter of height containing in the messages, and the checking of the height broadcasted in messages are not modelled in our specification. And we also only studied this kind of movement of the lift system: If the UP or DOWN button of one lift is pressed, all the lifts of the system should go up, respectively down. The initial specification for system with three lifts is given in Appendix I. Here we only highlight some parts of this specification. The part of data types is discussed in Section 4.2, the part of processes in Section 4.3.

4.2 Data types

Obviously we need to represent the physical structure of the lift system. This is done by means of the sort Address. The constructors of this data type consist of identifiers (one for each station). The functions suc and pre yield the identifiers of the neighbours in the circle. suc yields the one at the right side, pre yields the one at the left-hand side (see Figure 1). Because of the similarity in structure, we use this data type also to represent the position of a station. This data type is also used to identify the position of relays. Relay n is the one between the station with address n and the station with address suc(n); it is controlled by the station at the left side (addressed as n). We specify the sort Address with 3 elements below:

sort Address
func 1, 2, 3: →Address
map suc: Address→Address
pre: Address→Address
eq: Address×Address→Bool

[2]These two actions can also be represented as delta and tau in ASCII, respectively.

[3]In ASCII symbols, it is represented as $p < |b| > q$. We use both LaTeX and ASCII symbols in the following μCRL specifications to obtain a better layout. For example, we use $\sum_{\texttt{n:Address}}$ instead of **sum**(n:Address).

rew suc(1)=2 suc(2)=3 suc(3)=1
pre(1)=3 pre(2)=1 pre(3)=2
eq(1,1)=T eq(1,2)=F eq(1,3)=F
eq(2,1)=F eq(2,2)=T eq(2,3)=F
eq(3,1)=F eq(3,2)=F eq(3,3)=T

To model the bus, we must record which relays are closed. This is done by means of the sort `Alist`, which is a list of addresses. The constructors of this sort are `ema` and `set`. `ema` stands for an empty list. `set` constructs a new list by inserting an address into a list. The function `reset(a,A)` removes all the occurrences of the address `a` from the list `A`. Function `test(a,A)` tells us whether the address `a` is in list `A`. The function `empty(A)` is used to judge whether a list is empty, or not. `if(b,A,A')` is an auxiliary function to specify `test` and `reset`, where `b` is a data term of sort `Bool`. It is used to simulate conditional equations, meaning that if `b` holds then `A` is selected, otherwise `A'`. And the concatenation of two lists is represented by the function `conc(A,A')`. The function `Addresses(A,a)` is used to get the list of all stations connected to the station `a` via list `A` of closed relays. `a` is not included in the result (see in Appendix I).

sort Alist
func ema: →Alist
set: Address×Alist→Alist
map reset: Address×Alist→Alist
test: Address×Alist→Bool
empty: Alist→Bool
if: Bool×Alist×Alist→Alist
conc: Alist×Alist→Alist
Addresses: Alist×Address→Alist
var a, a': Address A, A': Alist
rew reset(a,ema)=ema
reset(a,set(a',A))=if(eq(a,a'),reset(a,A),set(a',reset(a,A)))
test(a,ema)=F
test(a,set(a',A))=if(eq(a,a'),T,test(a,A))
empty(ema)=T
empty(set(a,A))=F
if(T,A,A')=A
if(F,A,A')=A'
conc(ema,A)=A
conc(set(a,A),A')=set(a,conc(A,A'))

In our model, only the following states of stations are specified by a sort `State`: STANDBY, UP, DOWN, STARTUP and SYNC. The state SYNC is not really a state, but it can be broadcasted in a message instead of the states. This kind of message is used to synchronise the physical movement of all the lifts.

sort State
func STANDBY, UP, DOWN, STARTUP, SYNC: →`State`

The messages travelling on the network are specified by a sort `Message`. A message has the form `mes(m,s)`: `m` is the position of the station sending the message and `s` is the state of the sending station. By using the functions `getaddress` and `getstate`, we can get the position, respectively the state of the station.

sort Message
func mes: Address×State→Message
map getaddress: Message→Address
getstate: Message→State
var a: Address s: State
rew getaddress(mes(a,s))=a
getstate(mes(a,s))=s

4.3 Processes

In this section, we focus on the process part of our specification. It is obvious that we need to represent the behaviors of the bus and the station. The bus and the station are both modelled as processes.

The specification of the bus poses two problems. First, we must represent which relays are open and which ones are closed. This is done by parameterising the bus process with an `Alist R` of identifiers of all closed relays. If a station closes a relay, the identifier of the relay is added to this list. If it opens a relay, the identifier of the relay is removed from this list. This is achieved with the help of two actions `r_open-relay(n)` and `r_close-relay(n)`.

Second, we must represent the transportation of messages over the bus. In the system, a message put on the bus by one station is received by all the other stations connected to the sending station via closed relays. This is modelled by means of a delivery process (`Deliver`) parameterised with an Alist `A` of stations that have yet to receive the message. After accepting a message from a station with the action `r_stob(m,a)` (receive message `m` from station `a` to the bus), the bus process moves to the delivery phase, provided that the list `R` is not empty. This phase consists of a number of cycles. In each cycle, the message is delivered to one station in list `A` by the action `s_btos(m,a)` (send message `m` from the bus to station `a`) and then the next cycle is entered with the station `a` removed from list `A`. If the last station is removed, the bus process returns to the `Bus` phase. The `Deliver` process has `R` as one of its parameters; this is needed to restart the `Bus` process after the delivery phase with correct list of the closed relays. In the delivery phase, the bus does not accept messages from the stations, which ensures that a message broadcasted by a station is received by all other stations before the next station can send a message.

act r_stob, s_btos: Message×Address
r_open-relay, r_close-relay: Address
proc Bus(R:Alist) =
$\sum_{\texttt{mes:Message}} \sum_{\texttt{a:Address}}$
r_stob(mes,a)·
(Bus(R)
◁empty(Addresses(R,a))▷
Deliver(mes,R,Addresses(R,a))) +
$\sum_{\texttt{a:Address}}$
r_open-relay(a)·
Bus(reset(a,R)) +
$\sum_{\texttt{a:Address}}$
r_close-relay(a)·
Bus(set(a,R))

proc Deliver(mes:Message, R:Alist, A:Alist) =
$\sum_{\texttt{a:Address}}$
s_btos(mes,a)·
(Bus(R)
◁empty(reset(a,A))▷

Deliver(mes,R,reset(a,A)))
◁test(a,A)▷δ +
$\sum_{\texttt{a:Address}}$
r_open-relay(a)·
Deliver(mes,reset(a,R),A) +
$\sum_{\texttt{a:Address}}$
r_close-relay(a)·
Deliver(mes,set(a,R),A)

The actions `r_stob` and `s_btos` are intended to communicate with the actions `s_stob` (send a message from a station to the bus) and `r_btos` (receive a message from the bus to a station) into `c_stob` and `c_btos`, respectively. Likewise, the actions `r_open-relay` and `r_close-relay` are synchronised with the actions `s_open-relay` and `s_close-relay`.

comm s_stob | r_stob = c_stob
s_btos | r_btos = c_btos
s_open-relay | r_open-relay = c_open-relay
s_close-relay | r_close-relay = c_close-relay

After modelling the bus process, we come to the specification of the lift controller. The following actions are associated with the buttons of a lift. But they do not simply represent the physical action of pressing a button of the real system. Only those actions of pressing a button which have effect on the behavior of the system are modelled in our specification (see Figure 2). For example, in the normal operation phase, a SETREF button can be physically pressed. Because in this phase, a station does not respond to this action, the action `setref` cannot occur according to our specification of the normal operation phase (see the specification of `Lift2`). Since this, leaving out these actions does not affect our verification. The action of outputting state `s` of station `n` to the motor input is represented as the action `move(n,s)`.

act setref, up, down, released: Address
move: Address×State

The control of the lift system movement is divided into two phases. Initially, all relays are open. In the first phase (startup phase), the network connection is set up, and each station gets to know its position and the number of stations in the network. In the second phase (normal operation phase), the stations claim the bus in a fixed order and the physical movement of the system can be initiated. Each lift process is parameterised with an address `n`, which identifies the station.

The behavior of a station in the startup phase is modelled by two processes, `Lift0` and `Lift1`. Initially, all stations are in `Lift0`. `Lift0` specifies the initial behaviors of a station. In this phase, the SETREF button of a station can be pressed or a station can receive a STARTUP message from another one. `Lift1` models how the stations with a position greater than 1 get to know the number of stations in the network. The parameter `m` is added to `Lift1` to record the position of a station. The parameter `nos` is used to remember the number of the stations.

The station of which the SETREF button is pressed gets position 1. It closes its relay with the action `s_close-relay(n)` and broadcasts a STARTUP message. Next, it opens its relay with the action `s_open-relay(n)` and waits for a STARTUP message. When it gets the STARTUP message, it responds by changing its state to STANDBY and broadcasting its state, then it goes into the normal operation phase, which is modelled as `Lift2`. If a station (not the one, on which the SETREF button is pressed.) gets a STARTUP message, it adds 1 to the position of the message's sender and stores this both as its `m` and as its `nos`. It adapts the STARTUP state, closes its relay and broadcast its own state. Next, it

moves into `Lift1`, where it can change its own `nos` according to the position of the STARTUP message it receives. In the phase of `Lift1`, each station gets to know the number of the stations in the network by the position of the last received STARTUP message in the end. When a station with position greater than 1 gets a STANDBY message, it adopts its states to STANDBY and goes into process `Lift2` . If it is its turn to claim the bus, when it receives a message from its predecessor, it broadcasts a STANDBY message and then goes into process `Lift2`. In this way, the startup phase is finished and all stations are connected to one linear bus. The processes `Lift0` and `Lift1` are specified as follows:

proc Lift0(n:Address)=
setref(n)·
s_close-relay(n)·
s_stob(mes(1,STARTUP),n)·
s_open-relay(n)·
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(s_stob(mes(1,STANDBY),n)·
Lift2(n,1,getaddress(mes),STANDBY)
$\lhd$eq(getstate(mes),STARTUP)$\rhd\delta$) +
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(s_close-relay(n)·
s_stob(mes(suc(getaddress(mes)),STARTUP),n)·
Lift1(n,suc(getaddress(mes)),suc(getaddress(mes)))
$\lhd$eq(getstate(mes),STARTUP)$\rhd\delta$)

proc Lift1(n:Address, m:Address, nos:Address)=
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(Lift1(n,m,getaddress(mes))
$\lhd$eq(getstate(mes),STARTUP)$\rhd$
((s_stob(mes(1,STANDBY),n)·
Lift2(n,m,nos,STANDBY)
$\lhd$eq(getaddress(mes),pre(m))$\rhd$
Lift2(n,m,nos,STANDBY))
$\lhd$eq(getstate(mes),STANDBY)$\rhd\delta$))

Note that during the startup phase, all the stations expect to receive either a STARTUP message or a STANDBY message, otherwise it will result into a deadlock. This can be model checked later on.

The behavior of a station during the normal operation is specified by means of two processes (`Lift2` and `Lift3`). The parameter `s` is used to record the state of the station. In this phase, the stations broadcast their messages in a fixed order. A station knows that it is its turn to claim the bus when it receives a message from its predecessor. In both `Lift2` and `Lift3`, a station responds to an incoming SYNC message by immediately outputting its state to the motor input with the action `move(n,s)`. `Lift2` models the behavior of a station that is passive or in STANDBY. In this phase, a station will respond to a state message by adopting the state in the message. When a station gets the turn to claim the bus, it adopts the state in the received message and broadcasts it. In addition, a station in STANDBY will respond to an action of pressing a button. It adopts the corresponding state and becomes active (`Lift3`). `Lift3` models the behavior of an active station. The parameter `count` is used to count the number of stations that are in the same state as this active one. This counter is initiated with the number of stations in the network. Each time the active station receives a message with the same state as itself, the counter is decreased. When the active station gets the turn to use

the bus, it will determine whether it has received enough messages of the right type (i.e., whether its counter equals 2 and the state of the message of its predecessor is the same as the state of itself). If so, it will send a SYNC message, output its state to the motor, broadcast its own state and reset the counter to the number of the stations in the network. If not, it will broadcast its state and reset its counter. When the pressed button on the lift is released (modelled by `released(n)`), the active station returns to STANDBY.

```
proc  Lift2(n:Address, m:Address, nos:Address, s:State)=
      (up(n)·
      Lift3(n,m,nos,UP,nos)+
      down(n)·
      Lift3(n,m,nos,DOWN,nos))
      ◁eq(s,STANDBY)▷δ +
      ∑mes:Message
        r_btos(mes,n)·
          (move(n,s)·
          Lift2(n,m,nos,s)
          ◁eq(getstate(mes),SYNC)▷
            (s_stob(mes(m,getstate(mes)),n)·
            Lift2(n,m,nos,getstate(mes))
            ◁eq(getaddress(mes),pre(m))▷
            Lift2(n,m,nos,getstate(mes))))

proc  Lift3(n:Address, m:Address, nos:Address, s:State, count:Address)=
      released(n)·Lift2(n,m,nos,STANDBY)
      ◁not(eq(s,STANDBY))▷δ +
      ∑mes:Message
        r_btos(mes,n)·
          (move(n,s)·
          Lift3(n,m,nos,s,count)
          ◁eq(getstate(mes),SYNC)▷
              ((s_stob(mes(m,SYNC),n)·
              move(n,s)·
              s_stob(mes(m,s),n)·
              Lift3(n,m,nos,s,nos)
              ◁eq((getstate(mes),s)∧eq(count,2)▷
              s_stob(mes(m,s),n)·
              Lift3(n,m,nos,s,nos))
            ◁eq(getaddress(mes),pre(m))▷
              (Lift3(n,m,nos,s,pre(count))
              ◁eq(getstate(mes),s)▷
              Lift3(n,m,nos,s,count))))
```

By putting n `Lift0` processes and one `Bus` process in parallel, we model a system with n lifts ($n \geq 2$).

```
init  hide ({c_stob, c_btos, c_open-relay, c_close-relay},
        encap( {
          s_open-relay, r_open-relay,
          s_close-relay, r_close-relay,
          s_stob, r_stob, s_btos, r_btos}
```

Bus(ema) || Lift0(1) || Lift0(2) || ... Lift0(n)))

The encapsulation operator **encap** enforces the actions `s_open-relay, s_close-relay, s_btos and s_stob` to occur in communication with the actions `r_open-relay, r_close-relay, r_btos and r_btos`, respectively. To analyse the specification, all internal actions like the communication between bus and stations can be abstracted away, which is achieved by converting them into the τ action with the help of the **hide** operator.

5. Analysis results

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

It is interesting to see that the problems were being detected in a rather unordered fashion. For instance problem 1 showed itself by visualising the system behavior for a system with 3 lifts after hiding all communications to and from the bus and reducing the resulting transition system modulo branching bisimulation. The first sign of the problem was that not all internal actions had been removed. Trying to understand the reason for this uncovered the precise problem quickly. Note that trying to prove the requirements did not really play a role in detecting the problems. Their role is to be found in showing the absence of more problems.

Four errors were found in the original design. We discuss these problems 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 6).

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 (see Figure 1 and the example in 2.2). 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, station 1 sends the STANDBY message, which initiates normal operation, while the relay between station 1 and station 2 is opened, 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 result in a deadlock.

The developers had spotted this problem in the testing phase, but they were unaware of its cause. They had solved the problem by adding delays before sending a STARTUP message.

In our revised specification, the delay is modelled by the communication of two actions, `s_sync` and `r_sync`. This is enough to make us sure of only when station 2 waits till the relay between station 1 and station 2 is closed, it sends a STARTUP message[4].

Our experiments have indicated that this solves the problem adequately (if the delay is long enough to make sure that the relay between station 1 and station 2 is opened before station 2 sends the STARTUP message). The developers also indicate that it suffices to delay only the second STARTUP message. The main modification is made in the definition of process `Lift0`. It is shown together with the solution to the second problem at the end of Section 5.2.

5.2 Problem 2

The second problem occurs if the SETREF buttons of two lifts are pressed at almost 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 (see Figure 1). 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

[4]The operator **encap** can enforce the two actions `s_sync` and `r_sync` to occur in communication with each other, and not on their own.

conclusion that the circle is completed and initiates normal operation. At the same time D closes the relay between D and A and 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 neighboring lifts (say A and B) are pressed. Assume that B sends a STARTUP message 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, 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.

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. The process `Setref_monitor` is defined to prevent that in the startup phase more than one SETREF button is pressed at different lifts at the same time. The action `setref(n)` in `Lift0` is replaced by the action `s_init(n)`, which applies a lock on the monitor. After station 1 gets a STARTUP message, it releases the lock by the action `s_stable`. During the period when the monitor is locked, no other SETREF button pressed action can have effect on the whole lift system.

comm s_init | r_init = c_init
s_sync | r_sync = c_sync
s_stable | r_stable = c_stable

proc Setref_monitor =
$\sum_{\texttt{n:Address}}$
r_init(n)·
r_stable·
Setref_monitor

proc Lift0(n:Address)=
s_init(n)·
s_close-relay(n)·
s_stob(mes(1,STARTUP),n)·
s_open-relay(n)·
s_sync·
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(**s_stable**·
s_stob(mes(1,STANDBY),n)·
Lift2(n,1,getaddress(mes),STANDBY)
◁eq(getstate(mes),STARTUP)▷δ) +
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(s_close-relay(n)·
(**r_sync**·
s_stob(mes(2,STARTUP),n)·
Lift1(n,2,STARTUP)
◁eq(getaddress(mes),1)▷
s_stob(mes(suc(getaddress(mes)),STARTUP),n)·
Lift1(n,suc(getaddress(mes)),STARTUP))
◁eq(getstate(mes),STARTUP)▷δ)

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 avoid this problem in a very tricky way. It is useful for our continuous analysis. Given the chosen bus it seems impossible to solve this problem satisfactorily. As a result of our analysis, the implementation of the lift system was adapted. At initialization of the system, a random identifier is created to minimize the risk that more than one independent networks come into existence.

5.3 Problem 3

The third problem occurs if a button is pressed and released at an inappropriate 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 independently discovered by the developers when testing the system. The solution to this problem is simple. We 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. The main modification is made in the definition of process `Lift3`. It is shown together with the solution to the fourth problem at the end of Section 5.4.

5.4 Problem 4

The fourth problem occurs when two (UP or DOWN) 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 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. In revised specification, a `Bool` parameter is added into the definition of process `Lift2` to mark the station that wants to be active. It is set true when one button of the station is pressed. When it is the marked station's turn to use the bus, but it finds there is already an active station in the system, the marked station fails to be active. It adopts the state of the received message and broadcasts the message. Our experiments indicate that this solves the problem adequately.

```
proc  Lift2(n:Address, m:Address, nos:Address, s:State, c:Bool)=
      (up(n)·
```

Lift2(n,m,nos,UP,nos,**T**)+
down(n)·
Lift2(n,m,nos,DOWN,nos,**T**))
◁eq(s,STANDBY)▷δ +
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
(move(n,s)·
Lift2(n,m,nos,s,**c**)
◁eq(getstate(mes),SYNC)▷
(((s_stob(mes(m,s),n)·
Lift3(n,m,nos,s,nos)
◁ eq(getstate(mes),STANDBY)▷
s_stob(mes(m,getstate(mes)),n)·
Lift2(n,m,nos,getstate(mes),**F**))
◁ **c** ▷
s_stob(mes(m,getstate(mes)),n)·
Lift2(n,m,nos,getstate(mes),**F**))
◁eq(getaddress(mes),pre(m))▷
(Lift2(n,m,nos,s,**c**)
◁ **c** ▷
Lift2(n,m,nos,getstate(mes),**c**))))

proc Lift3(n:Address, m:Address, nos:Address, s:State, count:Address)=
released(n)·
Lift3(n,m,nos,STANDBY,nos)
◁not(eq(s,STANDBY))▷δ +
$\sum_{\texttt{mes:Message}}$
r_btos(mes,n)·
((s_stob(mes(m,STANDBY),n)·
Lift2(n,m,nos,STANDBY,**F**)
◁eq(s,STANDBY)▷
(s_stob(mes(m,SYNC),n)·
move(n,s)·
s_stob(mes(m,s),n)·
Lift3(n,m,nos,s,nos)
◁eq(getstate(mes),s)∧eq(count,2)▷
s_stob(mes(m,s),n)·
Lift3(n,m,nos,s,nos)))
◁eq(getaddress(mes),pre(m))▷
(Lift3(n,m,nos,s,pre(count))
◁eq(getstate(mes),s)▷
Lift3(n,m,nos,s,count)))

In addition, the developers also found that two stations have the same identifier when one of the relays is closed before startup phase. This ends up with that one station will do nothing, while the rest of the system carry on working. This phenomenon also can be revealed in our model. We simply parameterise the process `Bus` with a nonempty list of closed relays. The developers tackled this by generating an emergency stop. In our point of view of this system, this is not a problem with the protocol itself.

After these four problems are all repaired, no more problems have been found. We show by means

of model checking that that this modified specification meets the requirements in the next section. The specification for model checking is given in Appendix II.

6. Verification

6.1 Techniques and Input Language for Evaluator

Model checking is an automatic technique to determine whether a state transition system satisfies certain requirements [CGP00]. 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 be expressed as a temporal logic formula firstly. A model checker explores the reachable states of a certain state 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 transition system that violates the requirement.

The temporal logic used as input language for Evaluator [5] is called *regular alternation-free μ-calculus*. It is an extension of the alternation-free fragment of the modal μ-calculus with action predicates and regular expressions over action sequences. The *regular alternation-free μ-calculus* is built from three types of formulae, according to the syntax as follows [MS00]:

1. Action formulae $\alpha ::= a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2$
2. Regular formulae $\beta ::= \alpha \mid \beta_1 \cdot \beta_2 \mid \beta_1|\beta_2 \mid \beta^*$
3. State formulae $\varphi ::= \mathrm{F} \mid \mathrm{T} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid <\beta>\varphi \mid [\beta]\varphi \mid Y \mid \mu\mathrm{Y}.\varphi \mid \nu\mathrm{Y}.\varphi$

Action formulae α are built from action names $a \in A$ by the boolean operators. Regular formulae β are built from action formulae α by using the standard regular expression operators, such as concatenation ($\cdot$), choice ($|$) and transitive-reflexive closure (*). State formulae φ are built from proposition variables by using the standard boolean operators, the possibility $<\beta>\varphi$ and necessity operators $[\beta]\varphi$, and the least and greatest fixpoint operators $\mu Y.\varphi$ and $\nu Y.\varphi$. The intuitive meaning of the formula $<\beta>\varphi$ is "it is possible to make β-transition to a state where φ holds." Likewise, $[\beta]\varphi$ means that "φ holds in all states reachable by making a β-transition." The μ and ν are used to express least and greatest fixpoints, respectively. The boolean operators have the usual meaning: a state of the state transition system always satisfies T; it never satisfies F; it satisfies $\varphi_1 \vee \varphi_2$ if and only if it satisfies φ_1 or it satisfies φ_2; it satisfies $\varphi_1 \wedge \varphi_2$ if and only if it satisfies both φ_1 and φ_2.

6.2 Expressing requirements as formulae

There are five requirements on the lift system. The first property is a universal one: *deadlock freeness*. In the *regular alternation-free μ-calculus* syntax this is specified as follows:

P1 [T*]<T>T

stating that every state has at least one successor.

The second property is that of *Liveness I*, which means that buttons on the stations can eventually be pressed. The *regular alternation-free μ-calculus* code is given below, '.' is used to indicate the address of any lift:

P2.1 [T*]<T*·up(.)>T

P2.2 [T*]<T*·down(.)>T

It states that there exists a sequence leading to an UP or DOWN action after zero or more transitions.

The property of *Liveness II* is expressed in the *regular alternation-free μ-calculus* syntax below, we use '⋆' to indicate the address of a lift, on which the UP (or DOWN) button is pressed.

[5] It is a model checker in the Cæsar Aldébaran Development Package (CADP).

P3.1 [(¬(up(.) | down(.)))* · up(⋆) ·
(¬(up(.) | down(.) | released(⋆))))*]
<(¬(up(.) | down(.) | released(⋆)))* · move(.,UP)>T

P3.2 [(¬(up(.) | down(.)))* · down(⋆) ·
(¬(up(.) | down(.) | released(⋆))))*]
<(¬(up(.) | down(.) | released(⋆)))* · move(.,DOWN)>T

It says that in any execution sequence containing only one button-pressed action, and containing no button-released action of the pressed button, the whole system always begins to move.

The fourth property of our specification is *Safety I*. It says that if one of the lifts moves, all the other lifts should not move in the opposite direction. What is more, to keep the trucks in balance, all lifts have to move in the same direction simultaneously. To formalise this property, any order of the lifts' movements must be dealt with carefully. This means that the size of the formula grows in a factorial fashion with respect to the number of lifts.

To solve this problem, we split the formula into pieces which can be checked by the Evaluator. Taking a lift system with three stations as an example, one piece of this property in the *regular alternation-free μ-calculus* syntax is specified as follows:

P4 [normal_movement* ·
¬(move(1,UP) | move(2,UP) | move(3,UP))* ·
move(1,UP) ·
¬(move(1,UP) | move(2,UP) | move(3,UP))* ·
move(2,UP) ·
¬(move(1,UP) | move(2,UP) | move(3,UP))* ·
move(3,DOWN)
] F

The action predicate *normal_movement* denotes the sequence of the correct behaviors of the lift system. Above code says that in all paths consisting of normal movements of the system, lift 1 is the first to move up, after that, no movement of the other stations, and then lift 2 moving up, also no movements of other stations following; moreover, the action of lift 3 moving down always results in a state where F holds, equivalently, as long as lift 1 and lift 2 move up, lift 3 cannot move down. The other possibilities of the movement of stations can also be specified like this.

The fifth property of *Safety II* states that if no UP or DOWN button is pressed, then the system cannot move UP or DOWN. The following shows the code in *regular alternation-free μ-calculus*.

P5.1 [(¬ up(.))* · move(.,UP)] F

P5.2 [(¬ down(.))* · move(.,DOWN)] F

This should be read as follows: if an execution sequence does not contain button-pressed action, then in the resulting state the stations cannot move up or down.

A notorious problem when model checking is the *state space 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 transition system modulo weak equivalence [Mil89]. This allows an efficient minimization of the transition system space.

6.3 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. For any lift system consisting of six or more lifts, our toolset

Number of lifts	Number of states	Number of transitions	CPU time generation
2	383	716	2s11
3	7,282	18,957	11s80
4	128,901	419,108	3m54s95
5	2,155,576	8,676,815	1h32m54s70

Table 1: Transition system dimensions

fails to generate the transition system due to insufficient memory [6]. The size of transition system generated quickly increases with the number of the lifts. This dues to the buttons on each lift can be pressed in the arbitrary order.

The dimensions of the generated transition systems are summarised in Table 1. For each of the lift systems, the size of the generated transition system and the time it took to generate the system are given. Generation was performed on a 300 MHz SGI Origin 2000 R12000 Processor (8 Mb Cache) with 64 Gb memory.

7. Conclusion

In this paper, we have described a model of a distributed lift system. Our primary finding is that such a model is an efficient tool to understand the behavior of embedded distributed systems, in the sense that it helped us to find errors and understand their nature using the available technology. The developers of the system have fully (but initially reluctantly) acknowledged that these techniques have increased there understanding and are planning to release a new version of the product including the improvements we suggest. We also find confirmation of our previous findings that the possibility to describe interactions in a process algebraic way, and data using equational abstract data types provide exactly the required means for this specification and its validation. For detailed inspection, full descriptions of the lift system are added in the appendixes.

This case study also stressed the limitations of the μCRL toolset and interestingly enough, work as a catalyst to have its capacities enlarged. We have only been able to deal with systems with up to 5 lifts. For increased certainty, it would be nice to increase this number, preferably up to 32, as this is the maximal allowed configuration. Currently work is under way to generate and reduce transition systems on clusters of computers bringing the generation and manipulation of transition systems with billions of states within reach. However, extrapolation leads to the estimate that this is sufficient for only 6 to 7 lifts.

So, it is clear that more advanced techniques are needed, and much work into these is going on. It leads too far to mention all of them but work on parametric reduction of state spaces [GL01], confluence reduction [GS96] and parametric composition of parallel processes [GW01] are all activities striving to enable the analysis of systems with many more up to possibly unbounded parallel components.

8. Acknowledgements

We like to thank Wan Fokkink for comments on earlier versions of this report and Izak van Langevelde for help in model checking as well as comments on earlier versions. Thanks also go to the anonymous referees for correcting several English mistakes and suggesting many improvements.

[6] Instantiator: No memory to re-claim hashtable.

References

[BG96] B. Boigelot and P. 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.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[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.

[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*, 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.

[GL01] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, 2001.

[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. A balancing act: Analyzing a distributed lift system. In S. Gnesi and U. Ultes-Nitsche, editors, *Proceedings of the 6th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2001*, pages 1–12, 2001. The extended version also appeared as CWI technical report SEN-R0111, 2001.

[GS95] J.F. Groote and M.P.A Sellink. Confluence for process verification. In S.A. Smolka, editor, *Proceedings of CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 204–218. Springer-Verlag, 1995. Extended abstract appeared as [GS96].

[GS96] J.F. Groote and M.P.A Sellink. Confluence for process verification. *Theoretical Computer*

Science B (Logic, semantics and theory of programming), 170(1–2):47–81, 1996. Also appeared as [GS95].

[GW01] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. *Theoretical Computer Science*, 266(1-2):631–652, 2001. Also appeared as technical report CS-R9626 , Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 2000.

[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.

[MS00] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-cheching for regular alternation-free mu-calculus. In I. Schieferdecker and A. Rennoch, editors, *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany)*, April 2000.

[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.

[Wou01] A. G. Wouters. Manual for the μCRL toolset. Technical Report SEN-R0130, CWI, Amsterdam, 2001.

Appendix I
The initial μCRL specification of a system with 3 lifts

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mCRL specification of a distributed system for lifting trucks
% Arno Wouters and Pang Jun,  November 2000

% The parts of equality function of sorts are omitted to save space.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data type: Bool
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func T,F:->Bool
map  if:Bool#Bool#Bool->Bool
     not:Bool->Bool
     and:Bool#Bool->Bool
     or:Bool#Bool->Bool
var  b,b':Bool
rew  if(T,b,b')=b  if(F,b,b')=b'
     not(T)=F      not(F)=T     not(not(b))=b
     and(T,b)=b    and(F,b)=F
     and(b,T)=b    and(b,F)=F
     or(T,b)=T     or(F,b)=b
     or(b,T)=T     or(b,F)=b
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data type: State
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort State
func STANDBY:->State  % Stand by
     STARTUP:->State  % Start up (station is part of a not yet completely
                      % initialized network.)
     UP:->State       % Up   (lift is ready to go up.)
     DOWN:->State     % Down (lift is ready to go down.)
     SYNC:->State     % SYNC is not really a state, but it can be broadcasted
                      % in a message instead of the state.
```

```
    map  eq:State#State->Bool
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Data type: Address
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Address
    func 1,2,3:->Address           % There should be as many Addresses as there are lifts.
40  map  eq:Address#Address->Bool
         suc:Address->Address      % address of next station in network
                                   % successor modulo the number of lifts
         pre:Address->Address      % address of previous lift in network
                                   % predecessor modulo the number of lifts
    rew  suc(1)=2 suc(2)=3 suc(3)=1
         pre(2)=1 pre(3)=2 pre(1)=3
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Data type: Alist
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
50  sort Alist                     % A list of addresses (behaves as a multiset)
    func ema:->Alist               % empty address list
         set:Address#Alist->Alist
    map  reset:Address#Alist->Alist    % reset(a,A) - list A without address a
         test:Address#Alist->Bool      % test (a,A) - is address a in list A?
         empty:Alist->Bool             % empty(A)   - is list A empty?
         if:Bool#Alist#Alist->Alist    % select the first Alist if Bool is true
                                       % and the second Alist if Bool is false
         conc:Alist#Alist->Alist       % concatenate two lists
    var  a,a':Address
60       A,A':Alist
    rew  reset(a,ema)=ema
         reset(a,set(a',A))=if(eq(a,a'),reset(a,A),set(a',reset(a,A)))
         test(a,ema)=F
         test(a,set(a',A))=if(eq(a,a'),T,test(a,A))
         empty(ema)=T
         empty(set(a,A))=F
         if(T,A,A')=A
         if(F,A,A')=A'
         conc(ema,A)=A
70       conc(set(a,A),A')=set(a,conc(A,A'))
    % Addresses(A,a) is the list of stations connected to station a
    % via the relays in list A (a is not included).
    map  Addresses:Alist#Address->Alist
         Addresses-up:Alist#Address#Address->Alist
         Addresses-up-aux:Bool#Bool#Alist#Address#Address->Alist
         Addresses-down:Alist#Address#Address->Alist
         Addresses-down-aux:Bool#Bool#Alist#Address#Address->Alist
    var  a,a':Address
         A,A':Alist
80       b: Bool
    rew  Addresses(A,a)=conc(Addresses-up(A,a,a),Addresses-down(A,a,a))
         Addresses-up(A,a,a')=
         % if(test(a,A),set(suc(a),if(eq(suc(a),a'),ema,Addresses-up(A,suc(a),a'))),ema)
             Addresses-up-aux(test(a,A),eq(suc(a),a'),A,a,a')
         Addresses-up-aux(T,T,A,a,a')=set(suc(a),ema)
         Addresses-up-aux(T,F,A,a,a')=set(suc(a),Addresses-up(A,suc(a),a'))
         Addresses-up-aux(F,b,A,a,a')=ema
         Addresses-down(A,a,a')=
```

```
        % if(eq(pre(a),a'),ema,if(test(a,A),set(a,Addresses-down(A,pre(a),a')),ema))
            Addresses-down-aux(eq(pre(a),a'),test(pre(a),A),A,a,a')
        Addresses-down-aux(T,b,A,a,a')=ema
        Addresses-down-aux(F,T,A,a,a')=set(pre(a),Addresses-down(A,pre(a),a'))
        Addresses-down-aux(F,F,A,a,a')=ema
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Data type: Message
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Message
    func mes:Address#State->Message
    %    A message has the form: mes(m,s).
    %    m is the logical address of the station sending the message
    %    s is the state of the station sending the message
    map  getaddress:Message->Address % getaddress(m) - get the address parameter
         getstate:Message->State     % getstate(m)  - get the state parameter of
         eq: Message#Message->Bool   % whether two message are equal
    var  a,aa: Address  s,ss: State
    rew  getaddress(mes(a,s))=a
         getstate(mes(a,s))=s
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Processes: Bus and Deliver
    % Below the Bus is described. If it receives a message from a station
    % it will broadcast it to all stations connected via closed relays to the
    % original sender (but not to the sender itself).
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    act r_stob,s_btos:Message#Address
               % r_sob(m,a)  - receive message m from station a (station to bus)
               % s_btos(m,a) - send message m to station a (bus to station)
        r_open-relay,r_close-relay:Address
               % receive commands to open/close a relay

    proc Bus(R:Alist) =
         % R is the list of physical addresses of all closed relays.
         % receive a message and move to the delivery phase
         sum(m:Message, sum(a:Address,
             r_stob(m,a).
             (   Bus(R)
                 <| empty(Addresses(R,a))|>
                 Deliver(m,R,Addresses(R,a))
             )
         ))
         % receive a command to open a relay (i.e. add that relay to R)
           + sum(a:Address,r_open-relay(a).Bus(reset(a,R)))
         % receive a command to close a relay (i.e. remove that relay from R)
           + sum(a:Address,r_close-relay(a).Bus(set(a,R)))

         Deliver(m:Message,R:Alist,A:Alist)=
         % This is the phase of the bus where it delivers a message to all
         % connected stations. Note, that the bus is able to handle commands
         % to open or close relays even when it is busy delivering messages.

         % m - the message to be broadcasted
         % R - the list of all stations in the network
         % A - the list of all stations that have yet to receive m
         sum(a:Address,
```

```
            ( s_btos(m,a).
                (   Bus(R)
                    <| empty(reset(a,A)) |>
                    Deliver(m,R,reset(a,A))
                )
              <| test(a,A) |>
              delta
            )
        )
        + sum(a:Address,r_open-relay(a).Deliver(m,reset(a,R),A))
        + sum(a:Address,r_close-relay(a).Deliver(m,set(a,R),A))

    % The behavior of the station of each lift is described below.
    act s_stob,r_btos:Message#Address
            % s_stob(m,a) - send (to the bus) a package consisting of
            % message m and the address of the sending station (a) (station to bus)
            % r_btos(m,a) - receive (from the bus) a package consisting
            % of a message m and the intended receiver (a)
            % of this message (stations only accept packages
            % with their addresses as the destination) (bus to station)
        s_open-relay,s_close-relay:Address
            % by these two actions, the bus can add and remove this relay from the
            % the list of the closed relays.
            % The relay is the one between station (a) and its successor.
        % The following actions are associated with the buttons of a lift.
        setref:     Address   % setref button pressed at the start of the day
        up:         Address   % up button pressed
        down:       Address   % down button pressed
        released:   Address   % the button which was pressed is released
        % The following actions initiate hardware actions
        move:  Address#State  % output the sate to the motor of station
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Process: Lift0
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Lift0(n:Address)=
             setref(n).                 % setref button pressed
             s_close-relay(n).          % close its relay, between n and suc(n)
             s_stob(mes(1,STARTUP),n).  % send a STARTUP message
             s_open-relay(n).           % open its relay, between n and suc(n)
             sum(mes:Message,           % wait till it receives a message
                 r_btos(mes,n).
                 % if it is the expected message (start up).
                 %    (this message is supposed to originate from the last station
                 %      in the network)
                 (   s_stob(mes(1,STANDBY),n) .         % start broad casting
                     Lift2(n,1,getaddress(mes),STANDBY) % adopt STANDBY state
                     <| eq(getstate(mes),STARTUP) |>
                     delta
                 )
             )
        +
        sum(mes:Message,
            r_btos(mes,n).
            % if it is a startup message:
            (
```

```
                s_close-relay(n).              % close its relay
200             s_stob(mes(suc(getaddress(mes)),STARTUP),n).
                % and move to Lift1 (determine number of lifts)
                Lift1(n,suc(getaddress(mes)),suc(getaddress(mes)))
                <| eq(getstate(mes),STARTUP) |>
                delta
              )
        )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Lift1
% Lift1 catches all STARTUP messages, until a STANDBY message arrives.
210 % The logical address in the last STARTUP message is the number
% of stations in the network.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Lift1(n:Address,      % This station's identifier
           m:Address,      % This station's logical address (position number)
           nos:Address)    % Counts the number of stations
      =
       sum(mes:Message,
         r_btos(mes,n).
            % if it is a STARTUP message:
220         (
              % store its logical address and continue
              Lift1(n,m,getaddress(mes))
              <| eq(getstate(mes),STARTUP) |>
                % If it is a STANDBY message,
                % and the station get an STANDBY message
                % adopt the STANDBY state, and broadcast it
              ( ( s_stob(mes(m,STANDBY),n).Lift2(n,m,nos,STANDBY)
                  <| eq(getaddress(mes),pre(m)) |>
                  Lift2(n,m,nos,STANDBY)
230             )
               <| eq(getstate(mes),STANDBY) |>
               delta
               ) )
       )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Lift2
% Lift2 is normal operation.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Lift2(n:Address,       % This station's identifier
240        m:Address,       % This station's logical address (position)
           nos:Address,     % The number of stations in the network
           s:State          % The current state
          ) =
  (  up(n).Lift3(n,m,nos,UP,nos)
     + down(n).Lift3(n,m,nos,DOWN,nos)
  ) <| eq(s,STANDBY) |> delta
  + sum(mes:Message,
      r_btos(mes,n).
         (
250       % If it is a SYNC message
          move(n,s).Lift2(n,m,nos,s)
          <| eq(getstate(mes),SYNC) |>
          % other messages (other than SYNC):
```

```
                (
                % If it is this station's turn:
                % adopt the state in the message and broadcast it
                s_stob(mes(m,getstate(mes)),n).
                Lift2(n,m,nos,getstate(mes))
                <| eq(getaddress(mes),pre(m)) |>
                % If it isn't this station's turn:
                % just adopt the state in the message
                Lift2(n,m,nos,getstate(mes))
                )
            )
        )
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Process: Lift3
    % Lift3 is the state of an active lift (counting messages).
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Lift3(n:Address,       % This station's identifier
               m:Address,       % This station's logical address (position)
               nos:Address,     % The number of stations in the network
               s:State,         % The current state
               count:Address    % Counter
              ) =
      released(n).Lift2(n,m,nos,STANDBY)
      <| not(eq(s,STANDBY)) |>delta+
      sum(mes:Message,
          r_btos(mes,n).
          % If it is a SYNC message
          (
             move(n,s). Lift3(n,m,nos,s,count)
             <| eq(getstate(mes),SYNC) |>
             % other messages (other than SYNC):
             % if it is this station's turn
             (
                   % if the message is of the right type and the counter is right
                   (
                      s_stob(mes(m,SYNC),n).
                      move(n,s).s_stob(mes(m,s),n).
                      Lift3(n,m,nos,s,nos)
                      <| and(eq(getstate(mes),s),eq(count,2)) |>
                   % otherwise
                      s_stob(mes(m,s),n).Lift3(n,m,nos,s,nos)
                   )
             <| eq(getaddress(mes),pre(m)) |>
             % otherwise (not this station's turn)
                   % If the message is the one expected, decrease the counter.
                   (Lift3(n,m,nos,s,pre(count))
                    <| eq(getstate(mes),s) |>
                   % Otherwise
                   % do nothing
                   Lift3(n,m,nos,s,count))
             )
          )
        )

    act  c_stob, c_btos: Message # Address    % station to bus, resp. bus to station
```

```
        c_open-relay,c_close-relay:Address   % open/close relay

comm s_stob | r_stob = c_stob
     s_btos | r_btos = c_btos
     s_open-relay | r_open-relay = c_open-relay
     s_close-relay | r_close-relay = c_close-relay
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A system with 3 lifts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init
  hide({
        c_stob, c_btos,
        c_open-relay, c_close-relay
        },
      encap({
            s_stob, r_stob,
            s_btos, r_btos,
            s_open-relay, r_open-relay,
            s_close-relay, r_close-relay
            },
        Bus(ema) || Lift0(1) || Lift0(2) || Lift0(3)
            )
        )
```

Appendix II
The modified μCRL specification of a system with 3 lifts

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This is the "final" specification used for model checking.
% The data types and definition of process Bus are omitted,
% since they can be found in Appendix I.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act s_stob,r_btos:Message#Address
    s_open-relay,s_close-relay:Address
    s_init:     Address   % setref button pressed at the start of the day
    up:         Address   % up button pressed
    down:       Address   % down button pressed
    released:   Address   % the button which was pressed is released
    % The following actions initiate hardware actions
    move:  Address#State  % output the sate to the motor of station
    % synchronize actions used to prevent station 2 to send
    % a STARTUP message before the relay between 1 and 2 is opened
    s_sync r_sync
    % stable message used by the setref_monitor
    s_stable              % system ready for normal operation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Lift0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Lift0(n:Address)=
         s_init(n).                % setref button pressed
         s_close-relay(n).         % close its relay, between n and suc(n)
         s_stob(mes(1,STARTUP),n). % send a STARTUP message
         s_open-relay(n).          % open its relay, between n and suc(n)
         s_sync.                   % signal to station 2 that relay is open
         sum(mes:Message,          % wait till it receives a message
           r_btos(mes,n).
             % if it is the expected message (start up)
             % (this message is supposed to originate from the last station
             % in the network)
```

```
                (
                   s_stable.                              % stop two button monitor
                   s_stob(mes(1,STANDBY),n).              % start broad casting
                   Lift2(n,1,getaddress(mes),STANDBY,F) % adopt STANDBY state
                   <| eq(getstate(mes),STARTUP) |>
                   delta
                )
            )
            +
            sum(mes:Message,
                r_btos(mes,n).
                % if it is a startup message:
                (
                    s_close-relay(n).              % close its relay
                    % if this is station 2:
                    (
                        r_sync.                    % wait till relay between 1
                                                   % and 2 is closed
                        s_stob(mes(2,STARTUP),n).  % send start up message
                        % and move to Lift1 (determine number of lifts)
                        Lift1(n,suc(getaddress(mes)),suc(getaddress(mes)))
                        <| eq(getaddress(mes),1) |>
                        % if this is station 3 or higher:
                        % send STARTUP message
                        s_stob(mes(suc(getaddress(mes)),STARTUP),n).
                        % and move to Lift1 (determine number of lifts)
                        Lift1(n,suc(getaddress(mes)),suc(getaddress(mes)))
                    )
                    <| eq(getstate(mes),STARTUP) |>
                    delta
                )
            )
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Process: Lift1
    % Lift1 catches all STARTUP messages, until a STANDBY message arrives .
    % The logical address in the last STARTUP message is the number
    % of stations in the network.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Lift1(n:Address,      % This station's identifier
               m:Address,      % This station's logical address (position number)
               nos:Address)    % Counts the number of stations
          =
           sum(mes:Message,
               r_btos(mes,n).
                    % if it is a STARTUP message:
                (
                    % store its logical address and continue
                    Lift1(n,m,getaddress(mes))
                    <| eq(getstate(mes),STARTUP) |>
                    % If it is a STANDBY message,
                    % and the station get an STANDBY message
                    % adopt the STANDBY state, and broadcast it
                   (  (
                        s_stob(mes(m,STANDBY),n) . Lift2(n,m,nos,STANDBY,F)
                        <| eq(getaddress(mes),pre(m)) |>
```

```
                    Lift2(n,m,nos,STANDBY,F)
                  )
              <| eq(getstate(mes),STANDBY) |>
              delta
              )
            )
       )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process; Lift2
% Lift2 is normal operation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Lift2(n:Address,       % This station's identifier
           m:Address,       % This station's logical address (position)
           nos:Address,     % The number of stations in the network
           s:State,         % The current state
           c:Bool           % Button has been pressed on this station,
                            % and this station wants to be controller
       ) =
  (  up(n).Lift2(n,m,nos,UP,T)
     +down(n).Lift2(n,m,nos,DOWN,T)
  ) <| eq(s,STANDBY) |> delta
  +
  sum(mes:Message,
      r_btos(mes,n).
         (
           % If it is a SYNC message
           move(n,s).Lift2(n,m,nos,s,c)
           <| eq(getstate(mes),SYNC) |>
           % other messages (other than SYNC):
           % If it is this station's turn:
           (
              (% no more controllers, broadcast my state and begin counting
                 (
                   s_stob(mes(m,s),n).Lift3(n,m,nos,s,nos)
                   <| eq(getstate(mes),STANDBY) |>
                   % one controller before me, adopt the state and broadcast
                   s_stob(mes(m,getstate(mes)),n).Lift2(n,m,nos,getstate(mes),F)
                 )
              <| c |>
              % the station does not want to be  the controller,
              % adopt the state and broadcast it
                s_stob(mes(m,getstate(mes)),n).
                Lift2(n,m,nos,getstate(mes),c)
               )
            <| eq(getaddress(mes),pre(m)) |>
              % If it isn't this station's turn:
              % if the station wants to be controller
              (
                  %do nothing
                  Lift2(n,m,nos,s,c)
                  <| c |>
                  %otherwise
                  % adopts the state in the message
                  Lift2(n,m,nos,getstate(mes),c)
              )
```

```
            )
          )
    )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Lift3
% Lift3 is the state of an active lift (counting messages).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Lift3(n:Address,       % This station's identifier
           m:Address,       % This station's logical address (position)
           nos:Address,     % The number of stations in the network
           s:State,         % The current state
           count:Address    % Counter
          ) =
  released(n).Lift3(n,m,nos,STANDBY,nos) % after release the station waits for the
                                         % token to become passive
  <| not(eq(s,STANDBY)) |>delta+
  sum(mes:Message,
      r_btos(mes,n).
      % if it is this station's turn
      (
            (
            % if button was released
            % send STANDBY and become passive again.
            s_stob(mes(m,STANDBY),n).Lift2(n,m,nos,STANDBY,F)
            <| eq(s,STANDBY) |>
            % if button is still pressed
            % if the message is of the right type and the counter is right
                (
                % synchronize and move
                s_stob(mes(m,SYNC),n).
                move(n,s).s_stob(mes(m,s),n).
                Lift3(n,m,nos,s,nos)
                <| and(eq(getstate(mes),s),eq(count,2)) |>
                % otherwise, broadcast this station's state
                s_stob(mes(m,s),n).Lift3(n,m,nos,s,nos)
                )
            )
          <| eq(getaddress(mes),pre(m)) |>
          % If the message is the one expected, decrease the counter.
          (Lift3(n,m,nos,s,pre(count))
          <| eq(getstate(mes),s) |>
          % Otherwise
          % do nothing
          Lift3(n,m,nos,s,count))
      )
    )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Process: Setref_monitor
% Setref_monitor prevents that at the initial stage the setref button
% is pressed at more lifts at the same time.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act r_init: Address
    r_stable

proc Setref_monitor = sum(n:Address, r_init(n).r_stable.Setref_monitor)
```

```
200   act  c_stob, c_btos: Message # Address    % station to bus, resp. bus to station
           c_open-relay,c_close-relay:Address   % open/close relay
           c_sync                               % synchronize action to prevent that
                                                % message is send to fast
           c_init: Address         % actions needed to prevent two setref buttons to
           c_stable                % be pressed at the same time in the initial phase
      comm s_stob | r_stob = c_stob
           s_btos | r_btos = c_btos
           s_open-relay | r_open-relay = c_open-relay
           s_close-relay | r_close-relay = c_close-relay
210        s_sync | r_sync = c_sync
           s_init | r_init = c_init
           s_stable | r_stable = c_stable
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      %  A system with 3 lift3
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      init
        hide({
             c_stob, c_btos,
             c_open-relay, c_close-relay,
220          c_sync,
             c_init, c_stable
             },
           encap({
                s_stob, r_stob,
                s_btos, r_btos,
                s_open-relay, r_open-relay,
                s_close-relay, r_close-relay,
                s_sync, r_sync,
                s_init, r_init,
230             s_stable, r_stable
                },
             Bus(ema) || Lift0(1) || Lift0(2) || Lift0(3) || Setref_monitor
                )
             )
```