Analyzing the Redesign of a Distributed Lift System in UPPAAL [⋆]

Jun Pang[1], Bart Karstens[1] and Wan Fokkink[1,2]

[1] CWI, Department of Software Engineering,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands,
`{pangjun,bart,wan}@cwi.nl`
[2] Vrije Universiteit Amsterdam, Department of Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands, `wanf@cs.vu.nl`

Abstract. An existing distributed lift system was analyzed using the process algebraic language μCRL [7]. Four problems were found, three of which were also found independently by the developers in the testing phase. They solved these problems in an *ad hoc* manner, because the causes of the problems were unclear. The analysis in [7] revealed the reasons for those problems, and proposed solutions.
In this paper, we checked the developers' solutions using Uppaal. We show that the solutions of the developers do not solve these problems completely, while a refined version of our solution proposed in [7] does.

1 Introduction

As is well known, distributed algorithms form a major aspect of system design. Verifying the correctness of the protocols that regulate the behavior of distributed systems is usually a formidable task, as even simple behaviors become wildly complicated when they are carried out in parallel. Formal verification is a suitable approach to check whether a system meets its requirements.

In a formal model of a real-life system, details irrelevant to the requirements under scrutiny can be abstracted away. With the formal model at hand, one is able to reason about the system in a systematic and automatic way, using for example a model checker or a theorem prover. This formal reasoning can detect errors and suggest ways in which the system can be improved or optimized. A model is never completely equal to the original system, because it describes the system at a certain level of abstraction. This means that we can never be hundred percent sure that the system is correct with respect to the checked requirements. To achieve more confidence with the verified system, the model can be refined by adding more details. In this paper, we report some experience related to this topic by analyzing the redesign of a distributed lift system.

This lift system is used in real life for lifting lorries, railway carriages, buses etc. A system consists of a number of lifts: each wheel is supported by one lift

[⋆] This research is partly supported by the Dutch Technology Foundation STW under the project CES5008: Improving the quality of embedded systems using formal design and systematic testing.

and each lift has its own micro controller. This system has been designed and implemented by a small Dutch company (for commercial reasons we are not at liberty to reveal the company name). A special protocol has been developed to let the lifts operate synchronously. When testing their implementation the developers found three problems, but the causes of two of them were unclear. They solved these problems in an *ad hoc* manner. In order to explain the reasons and to make sure there are no more errors, the lift system was specified and verified in μCRL [8] and its toolset [5] in close cooperation with the developers. The three problems that were found by the developers were also found in the μCRL model. This indicated that the specification is actually close to the implementation. Another new problem was found in the model, which is indeed present in the system. The causes for the problems were detected, and solutions were proposed and included in the μCRL specification. The modified μCRL specification was shown to satisfy all the requirements by model checking. However, this happened independently of the developers, who decided not to wait for the results of the formal analysis and to redesign their implementation based on their own solutions. To distinguish between the two lift systems in this paper, we call the first lift system 'original design' and the one with the solutions of the developers 'redesign'.

The developers experienced a new problem in the redesign. Again the reason was unclear. Since the error traces displayed a regular pattern in time, the developers thought modeling exact timing might reveal the reason for this problem. In the μCRL specification, time is abstracted away. We could extend the μCRL model with exact timing information, but there is no automated verification toolset for timed process algebras. Therefore it was decided to use Uppaal [11], which is a toolset for validation and model checking of real time systems.

The Uppaal model of the redesign is achieved in several steps. First the μCRL model is translated into Uppaal. Then the Uppaal model is refined to move it closer to the real system; each lift is split into two components, where one component communicates with the other lifts and the other component can receive input from the environment. The developers' solutions for the aforementioned problems are adopted. After discussions with the developers, exact timing information is added. The requirements for the lift system are formulated in Uppaal, using its requirement specification language and test automata, and model checked. Using the graphic simulation tool in Uppaal, we detect the reason for the new problem, which the developers encountered in the redesign. We propose a new solution, which is based on the solution that was already put forward in [7]. The Uppaal model with the new solution satisfies all the requirements.

The developers acknowledge the efficiency and usefulness of formal verification for their redesign. Our solution will be implemented in the new release of the lift system; they are now more confident in the correct functioning of the redesigned lift system. The developers stress that formal methods should be applied in the early design phases to save testing effort and cost.

2 The lift system

2.1 Layout of the lift system

The lift system consists of an arbitrary number of lifts. Each lift supports one wheel of a vehicle. Different lift systems may have a different number of lifts, but this has no influence on the analysis, since this network should operate in the same way regardless how many lifts are connected.

Every lift has its own buttons. Three buttons are taken into account in the model: UP, DOWN and SETREF. If an UP or DOWN button on a certain lift is pressed, all lifts in the system should move up or down. Pressing a SETREF button on a lift is the only way a run of the system can start.

The movement of a lift system is controlled by means of a micro controller. Each lift has its own micro controller, called station. The stations can adopt four different states: STARTUP, STANDBY, UP and DOWN. The state of a lift can change in two ways: when a button on the lift is pressed, or by receiving a message from the network.

In the lift system, the data field of the messages transferred over the bus can contain two pieces of information: the position of the sender station, and the type of the message. There are two types of messages: SYNC messages and state messages. State messages broadcast the state of the sending station to the other stations. SYNC messages initiate physical movement. In response to a SYNC message, each station will immediately transfer its state to the motor of the lift, which causes movement. If the station is in UP, the lift will move up a fixed distance; if it is in DOWN, the lift will move down.

All the stations are connected to a CAN (Controller Area Network) bus [6]. The CAN bus is a simple, low-cost, multi-master serial bus with error detection capabilities. The bus transmits messages to the stations. Whenever a station wants to send a message, it is said to claim the bus. Stations can receive messages at any moment, but when a station wants to send a message it has to wait until it is its turn to use the bus. In the CAN bus, all stations can claim the bus at each cycle and several stations can claim the bus simultaneously. A non-destructive arbitration mechanism is used to determine which station may send its message. The resulting usage of the bus is ordered, and the stations take fixed turns to send their messages. To achieve this orderly usage of the bus, before the real use of the lift system we call 'normal operation', a start-up phase has been designed. In this phase each station finds out its position in the network and the total number of lifts in the network. When each station has been assigned a unique position, a virtual token can pass among the stations in the same order cycle after cycle. A station knows whether it is its turn to use the bus by checking the position of the sender station in the received message. The orderly usage of the bus during normal operation plays a crucial role in the analysis of the requirements and in the problems the lift system faces.

Control of the lift: Start-up The start-up phase has two functions. First it assigns a unique position to each lift in the network. This position works as an identity. When each lift has got its own position, an orderly usage of the bus is possible. 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. In order to do this, each station must know how many stations there are in the network.

There is a relay between every pair of adjacent stations and each relay is controlled by the station at its left side. When the system is switched on all the relays are open.

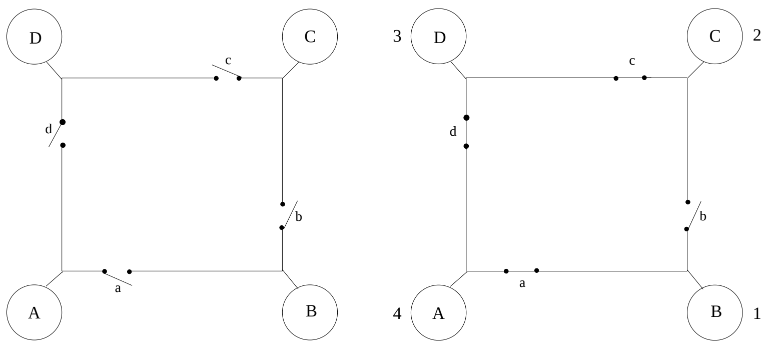

Fig. 1. State of the relays before (left) and after (right) initialization

The start-up phase is initialized by the station where the SETREF button is pressed. This station will behave differently from the other stations in the network. It will act as follows (chronological order):

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 other stations receive a STARTUP message from another station. The first time a station receives a STARTUP message, it will act as follows:

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 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 that in the left part of Fig. 1, the SETREF button of station B is pressed. The end result is that all stations are connected in the manner pictured in the right part of Fig. 1. All stations know their position and all stations know that there are four lifts in the network. More explanation about the start-up phase can be found in [7].

Control of the lift: Normal Operation When the start-up phase is finished, all the stations are in STANDBY. During the normal operation phase, the first station broadcasts its state, then the next station broadcasts its state and so on, until the last station has broadcast its state, after which the first station starts again. The state of a lift is changed if its UP or DOWN button is pressed. The station where this happens is called an active station. The active station will send an UP or DOWN message, according to the button that was pressed at the station. Passive stations change their state according to the messages they receive, and when it is their turn to use the bus they broadcast a message according to their state. These messages are received by all the other stations, and the active station is the only one that will count them. When it counts enough state messages, the active station will send a SYNC message, after which all the lifts move. The ordered sending of messages makes sure that the active station counts no more than one message from each station. In contrast to the passive stations, the state of the active station can only change if the pressed button is released again. In that case its state changes to STANDBY and the station becomes passive again. More details about this phase, including what happens when two UP or DOWN buttons at different lifts are pressed at the same time, will be discussed in Section 3.

2.2 Requirements

The desired behavior of the system is formulated in five requirements it has to fulfill. These requirements are listed below:

1. Deadlock freeness: The 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 UP or DOWN 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.

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 was pressed. The lifts will not move if no one has pressed an UP or DOWN button.

The two liveness requirements make sure that buttons can always be pressed and in response the lifts will always move. The two safety requirements make sure that the system will move properly. In Section 3, we present four problems in the original lift system. If the lift system satisfies the five requirements above, those four problems are guaranteed to be resolved.

3 UPPAAL model of the redesign

UPPAAL [11] is a toolset for validation and model checking of real time systems, which are modeled as networks of timed automata [2] extended with global shared variables. It consists of a number of tools including a graphic editor for system description, a simulator and a model checker. The idea of the UPPAAL toolset is to model a system using timed automata, simulate it and then verify properties of the system. During the design phase, the graphic simulator is used intensively to validate the dynamic behavior of each design sketch, in particular for fault detection, and later on for debugging the generated diagnostic traces. The verifier mainly checks for invariants and reachability properties. It does so by exploring the state space of a system using 'on the fly' searching techniques. It uses symbolic techniques to reduce the verification of modal logic formulas to solving simple reachability constraints. Some notable recent case studies with UPPAAL are [9, 12, 3].

The UPPAAL model presented in this section is the result of a few steps. First the μCRL model of the original design is translated into UPPAAL. This model is then changed into a representation of the redesign by adding the developers' solutions to the problems, that were found in the original design. The UPPAAL model of the redesign is also more specific, since interactions between the environment and the lift system are added that were abstracted away in the μCRL model of the original design. Furthermore, the model is extended with exact timing information. With respect to the explanation of the original design in Section 2, the redesign can be viewed as a refinement of the μCRL model. However, the desired behavior of the lift is basically the same as explained in Section 2. The redesign should therefore meet the same requirements as the original design.

The UPPAAL model contains four components. They are automata: *Station*, *Bus*, *Interface* and *Timer*. In UPPAAL, an automaton can be instantiated an arbitrary number of times. As explained in Section 2, the lift system consists of one bus and an arbitrary number of lifts. The automaton *Bus* models the CAN bus. For each lift in the system, we create two automata: *Station* and *Interface*. The automaton *Station* models the micro controller. In automaton *Interface*, the pressing and releasing of buttons on the lift is modeled. The automaton *Timer* is used to model time delay. In this section we will walk through the model. Due to space limitation, pictures of these automata are presented with only superficial explanation. Detailed information can be found in [10].

3.1 Transforming the μCRL model

The original lift system has been analyzed in μCRL [8], which combines the process algebra ACP [4] with equational abstract data types. To analyze the redesign of this system, first we transform the μCRL model into Uppaal. In this section, we discuss some model choices that have been made.

Value passing The μCRL specification of a process is constructed from action names, recursion variables and process algebraic operators. Actions and recursion variables carry zero or more data parameters. Parallel composition $p \parallel q$ interleaves the actions of processes p and q; moreover, actions from p and q may also synchronize to a communication action, when this is explicitly allowed by a predefined communication function. Two actions can only synchronize if they occur at the same time, and if their data parameters are semantically the same, which means that communication can be used to represent data transfer from one process to another. The communication function was used heavily in the μCRL specification in [7] to model the communications between the bus and stations. However in Uppaal, data transfer (or value passing) between processes (or automata) cannot be modeled in this way.

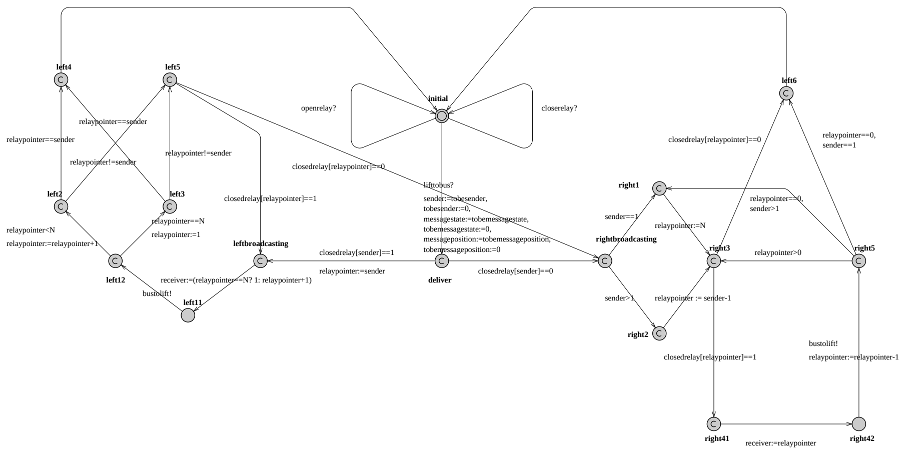

Fig. 2. The automaton *Bus*

We define two channels between the bus and stations: *bustolift* and *lifttobus*, and declare several global variables for data transfer when communication happens. When a station wants to send a message to the bus, it has to instantiate the values for some global variables in the message, for instance the state and the sender's position. When communication takes place, the values of those global variables are saved to the variables used by the bus. After communication, those global variables are provided with default values. In a similar fashion, messages are sent from the bus to stations. Detailed information can be found in the automata *Station* and *Bus* (see Fig. 2, Fig. 4 and Fig. 5).

Messages broadcasting In μCRL, summation $\sum_{d:D} p(d)$ provides the possibly infinite choice over a data type D. In the μCRL specification of the bus, when the bus gets a message from a station, it can compute the set of stations who can get this message via closed relays. Then the bus can choose one station from the set nondeterministically, and send it the message. By this way, we can model the broadcasting of a message. In UPPAAL, the summation operator is absent. We set a kind of fix order for the bus to broadcast a message. The relay controlled by a station is modeled as a flag. When the relay is closed, the flag is set to 1; otherwise it is 0. When a bus broadcasts a message, it starts to check the flag at the position of the message sender. If the flag is 1, it sends a message to the station connected by this relay, and continues to check the flag of this station. As soon as it reaches a flag with value 0, it continues at the station preceding the message sender. If the flag at this station is 1, the message is sent to the station, and the bus continues to check the flag at the preceding station. This procedure moves on until the bus reaches another flag with value 0. Recall that in both phases of the lift system, there is at least one open relay, which guarantees that the broadcasting procedure terminates. In the automaton *Bus* (see Fig. 2), when a bus gets a message at the 'initial' node, it starts broadcasting the message from the left part of the picture, then continues at the right part, and finally goes back to the 'initial' node.

One SETREF button pressed In [7], the second problem of the original design was found during the start-up phase. It occurs if the SETREF buttons at two lifts are pressed. The result of the problem is that after the start-up phase there will be two lift systems instead of one. The situation may lead to the violation of all the requirements. Given the chosen bus it seems impossible to solve this problem satisfactorily. The developers chose to emphasize in the manual that it is important to make sure that in the start-up phase the SETREF button of only one lift is pressed. We also take this assumption into our analysis of the redesign.

In the UPPAAL model it is impossible to press another SETREF button after one is pressed. We use guards on transitions to block pressing of SETREF buttons after one SETREF button has been pressed. In the automaton *Interface* (see Fig. 3), a variable *onesetref* is used as a guard on both transitions from the initial state. Initially the variable is zero, so one *Interface* can take the transition with the guard 'onesetref==0', if the SETREF button on the lift is pressed. The variable *onesetref* is now set to 1. In order to leave their initial state, the other *Interface* automata have to take the other transition with the

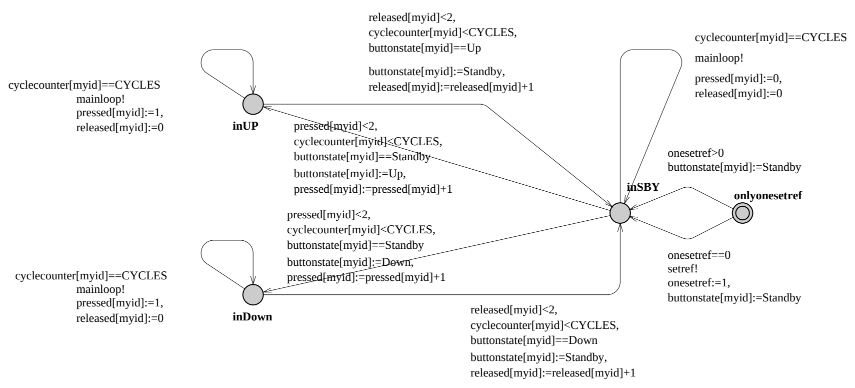
released[myid]<2,
cyclecounter[myid]<CYCLES,
buttonstate[myid]==Up
buttonstate[myid]:=Standby,
released[myid]:=released[myid]+1
cyclecounter[myid]==CYCLES
mainloop!
pressed[myid]:=0,
released[myid]:=0
cyclecounter[myid]==CYCLES
mainloop!
pressed[myid]:=1,
released[myid]:=0
inUP
pressed[myid]<2,
cyclecounter[myid]<CYCLES,
buttonstate[myid]==Standby
buttonstate[myid]:=Up,
pressed[myid]:=pressed[myid]+1
onesetref>0
buttonstate[myid]:=Standby
inSBY
onlyonesetref
pressed[myid]<2,
cyclecounter[myid]<CYCLES,
buttonstate[myid]==Standby
buttonstate[myid]:=Down,
pressed[myid]:=pressed[myid]+1
onesetref==0
setref!
onesetref:=1,
buttonstate[myid]:=Standby
cyclecounter[myid]==CYCLES
mainloop!
pressed[myid]:=1,
released[myid]:=0
inDown
released[myid]<2,
cyclecounter[myid]<CYCLES,
buttonstate[myid]==Down
buttonstate[myid]:=Standby,
released[myid]:=released[myid]+1

Fig. 3. The automaton *Interface*

guard 'onesetref>0'. Therefore it is simply made impossible to press more than one SETREF button in our UPPAAL model.

3.2 Adding the solutions

In the automaton *Station*, the two phases of the lift system as explained in Section 2 are clearly distinguishable.

Start-up Until all the stations have reached the node 'normaloperation', it is in the start-up phase. The main role of the start-up phase is to find out which position a lift has in the network and how many lifts there are in the network. The variables *position* and *number* are assigned to each lift to store this information.

The station where the SETREF button is pressed will move clockwise in Fig. 4 from the 'initial' node. It gets position 1, closes its relay, and sends a STARTUP message to the bus. After that it opens its relay and waits for a STARTUP message. When it gets the STARTUP message, it adopts the value of the variable *number* in this message; this way it gets to know how many lifts there are in the system. Then, it sends a STANDBY message and reaches the 'normaloperation' node. The other stations will move anti-clockwise in Fig. 4 from the 'initial' node. They first get a STARTUP message, increase the sender of the message by one, and save it as their own *position.* They close their own relay and send a STARTUP message. There is a small loop in Fig. 4, to indicate that the stations keep getting STARTUP messages and changing the knowledge of the number of lifts in the system. In the end, they will get a STANDBY message, and end up in the 'normaloperation' node. When all the stations have reached the 'normaloperation' node, all the stations are STANDBY. They all have a unique value for *position*, and the value of *number* of all the lifts is equal to the total number of lifts in the network.

Some time delays are added into the start-up phase to solve one problem found during testing. The timing information will be discussed in Section 3.3.

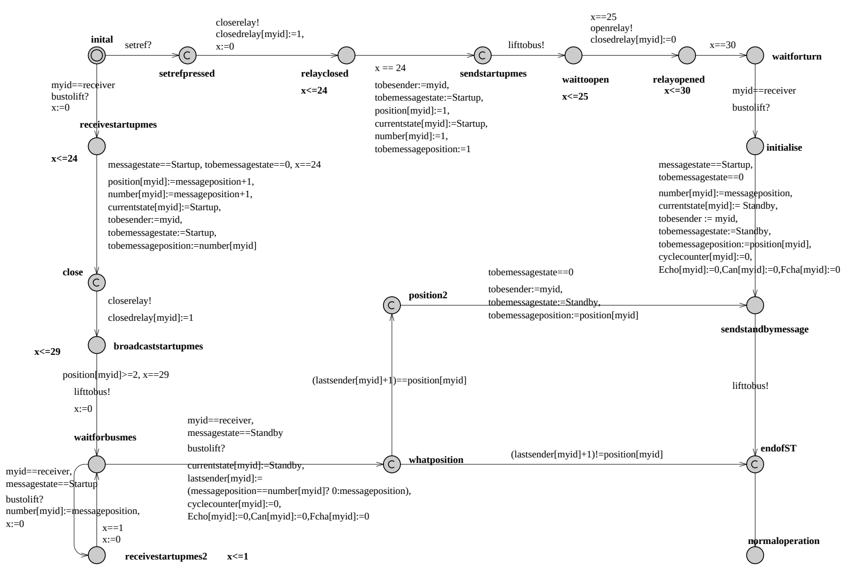

Fig. 4. The automaton *Station*: Start-up phase

Normal operation At node 'normaloperation', a station enters the normal operation phase, which is depicted in Fig. 5. In the normal operation phase, a distinction is made between two loops which a station can perform. One is the 'main loop', which takes place at the node 'normaloperation' in Fig. 5; and the other one we will call 'internal loop', which is the other part of Fig. 5. The difference between the main loop and the internal loop can be stated as follows: in a main loop the station receives state messages from its *Interface* and can change its state accordingly, and in an internal loop the station exchanges state messages with *Bus* and changes its state accordingly.

The main loop is a short loop in which the automaton *Station* synchronizes with its *Interface*. Executing the main loop is the only way the station can get information about which button on the lift (if any) is pressed or released. This main loop takes place after a fixed number of internal loops, which is modeled as a constant *CYCLES* in the Uppaal model. And a counter *cyclecounter* is used to record the number of internal loops that have happened after the last main loop. When 'cyclecounter==CYCLES', the main loop takes place and *cyclecounter* is reset to 0. If the station detects a difference between its current state (modeled by variable *currentstate*) and the state of the *Interface* (modeled by variable *buttonstate*), the station may change its state and adopt the one from the *Interface*. The main loop is also part of the original design, but it was abstracted away in the μCRL model in [7]. In the Uppaal model of the redesign it could not be left out, because as we will see the solutions from the developers interact in a critical way with the main loop.

In an internal loop, a station can do several things. First a station can get messages from the bus. Second, a station can send a message to the other stations, if it gets the turn to use the bus. Third, the active station can count state messages and initiate a movement of the whole system. In that case the active station will enter the node 'activemovement', while the other stations get a SYNC message and enter the node 'passivemovement'. A variable *move* is associated to each station to indicate the direction of the current movement.

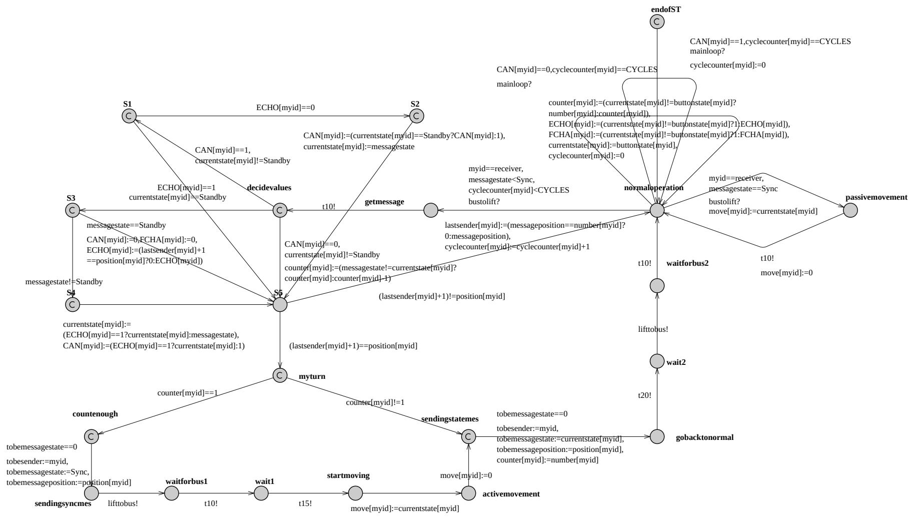
endofST
CAN[myid]==1,cyclecounter[myid]==CYCLES
mainloop?
cyclecounter[myid]:=0
CAN[myid]==0,cyclecounter[myid]==CYCLES
mainloop?
counter[myid]:=(currentstate[myid]!=buttonstate[myid]?
number[myid]:counter[myid]),
ECHO[myid]:=(currentstate[myid]!=buttonstate[myid]?1:ECHO[myid]),
FCHA[myid]:=(currentstate[myid]!=buttonstate[myid]?1:FCHA[myid]),
currentstate[myid]:=buttonstate[myid],
cyclecounter[myid]:=0
S1
ECHO[myid]==0
S2
CAN[myid]:=(currentstate[myid]==Standby?CAN[myid]:1),
currentstate[myid]:=messagestate
CAN[myid]==1,
currentstate[myid]!=Standby
myid==receiver,
messagestate<Sync,
cyclecounter[myid]<CYCLES
bustolift?
normaloperation
myid==receiver,
messagestate==Sync
bustolift?
move[myid]:=currentstate[myid]
passivemovement
ECHO[myid]==1
currentstate[myid]==Standby
decidevalues
S3
t10!
getmessage
messagestate==Standby
CAN[myid]:=0,FCHA[myid]:=0,
ECHO[myid]:=(lastsender[myid]+1
==position[myid]?0:ECHO[myid])
lastsender[myid]:=(messageposition==number[myid]?
0:messageposition),
cyclecounter[myid]:=cyclecounter[myid]+1
CAN[myid]==0,
currentstate[myid]!=Standby
counter[myid]:=(messagestate!=currentstate[myid]?
counter[myid]:counter[myid]-1)
t10!
waitforbus2
t10!
move[myid]:=0
messagestate!=Standby
S4
S5
(lastsender[myid]+1)!=position[myid]
currentstate[myid]:=
(ECHO[myid]==1?currentstate[myid]:messagestate),
CAN[myid]:=(ECHO[myid]==1?currentstate[myid]:1)
lifttobus!
(lastsender[myid]+1)==position[myid]
wait2
myturn
counter[myid]==1
t20!
countenough
counter[myid]!=1
sendingstatemes
tobemessagestate==0
tobesender:=myid,
tobemessagestate:=currentstate[myid],
tobemessageposition:=position[myid],
counter[myid]:=number[myid]
gobacktonormal
tobemessagestate==0
tobesender:=myid,
tobemessagestate:=Sync,
tobemessageposition:=position[myid]
waitforbus1
wait1
startmoving
move[myid]:=0
activemovement
sendingsyncmes
lifttobus!
t10!
t15!
move[myid]:=currentstate[myid]

Fig. 5. The automaton *Station*: Normal operation

Flags Problem three and four found in [7] occur in the normal operation phase. The third problem happens when an UP or DOWN button is pressed and released at an inappropriate moment. The lift system will end up in the situation that all stations are in UP or DOWN state, but there is no active station. This means that all the lifts will remain in that state until the system is shut down. This problem violates property Liveness II in Section 2.2. The reason for this problem is that in the original system a station becomes passive as soon as the pressed button on this lift is released. This problem was discovered by the developers when testing the system, and they solved it by means of flags.

The fourth problem occurs when two UP or DOWN buttons on different lifts are pressed at the same time and one of them is released at an inappropriate moment. As a result, some lifts will move, and one lift (where the button is released) remains at the same height. This violates property Safety I in Section 2.2. The reason for this problem is that a station becomes active as soon as a button on this lift is pressed. This problem was unknown to the developers and found its way into the final implementation of the original system. The detailed description of each problem can be found in [7]. We proposed to solve this problem by allowing a station to decide to be active or passive only when it is its turn to use the bus. In this paper, we focus on the solutions from the developers, and explain how they fail to solve the problems in Section 4. Furthermore, in Section 5 we refine our solution from [7], and show that it does solve the problems.

The developers attempted to solve the third problem with flags. When they are set their value is 1, and when they are reset their value is 0. The flags serve as blocks: they can prevent state changes when they are set. Two type of flags are used in the redesign, i.e. CAN, ECHO. Every station has its own flags. Initially all flags are 0. The CAN flag is set when a station receives a state message from the bus. An exception is the STANDBY message. If a station receives this message, the opposite happens: CAN is reset, but only when the current state of the station is also STANDBY; otherwise CAN is left unchanged. The idea of the developers was to use the CAN flag to block state changes by the main loop. If CAN is set, the main loop cannot change the state of the station. In Fig. 5, we have two main loops with different guards. One is 'CAN==1', and the other 'CAN==0'. If 'CAN==0' the main loop is taken. The current state of the station is compared with the *Interface*. In Fig. 3, *Interface* can communicate with *Station* when it is in the nodes 'inUp' (the UP button is pressed), 'inDown' (the DOWN button is pressed) or 'inSby' (no button is pressed). If 'CAN==1', some counters such as *cyclecounter* are reset, but nothing else happens.

The ECHO flag can only be set via the main loop with guard 'CAN==0'. When the station detects a difference between its current state and the state of the button, ECHO is set. When ECHO is set, the state of the station cannot change by messages it receives from the bus. Like CAN, ECHO can only be reset when the state of the station is STANDBY and a STANDBY message is received from the bus. But for ECHO, there is an extra requirement that has to be fulfilled before it can be reset: it has to be the station's turn to use the bus.

3.3 Adding timing information

The time model in UPPAAL is continuous or dense. Clocks are used to capture time in UPPAAL. They can be associated with a transition or a node. In a transition, clock variables can be reset or used as a guard. In a node, clock variables can be used as a hold up to let the process stay in that node for a certain amount of time. Such nodes are said to be labeled with an invariant.

The way we modeled the time information of the lift system is influenced by the developers' solution to solve one problem found in the start-up phase. It is also influenced by the fact that during normal operation the stations take fixed turns to use the bus. During the start-up phase there is no such order. This difference has led to a different treatment of the timing information in the two phases. We first discuss the start-up phase and then normal operation.

Start-up The first problem found in [7] occurs in the start-up phase. It has to do with the re-opening of the relay between the first and second lift at the wrong moment. Consider Fig. 1 in Section 2 again. The SETREF button is pressed on station B, which closes its relay and sends a STARTUP message to station C. If station C sends a STARTUP message before the relay between station B and station C is opened, this message is received by station B, which draws the incorrect conclusion that there are only two lifts in the network.

The solution to this problem is to let station C (or in general the station with position 2) wait until the relay between the first station and the second station is opened, before sending the STARTUP message. The developers added delays to the original design to make sure this happens.

In the redesign, during the start-up phase, a local clock 'x' is assigned to each station. The local clock is reset when a station gets a STARTUP message, or a SETREF button is pressed. This is used to capture the moment when the stations join the network. Receiving a message from the bus or sending a message to the bus costs 1 millisecond. The opening and closing of a relay cost 5 milliseconds. There is a delay of 24 milliseconds before sending a STARTUP message. This is all the timing information in the start-up phase.

Normal operation During normal operation, the local clocks used during the start-up phase are not used anymore. Instead we use one global. We create an extra automaton *Timer* depicted in Fig. 6.

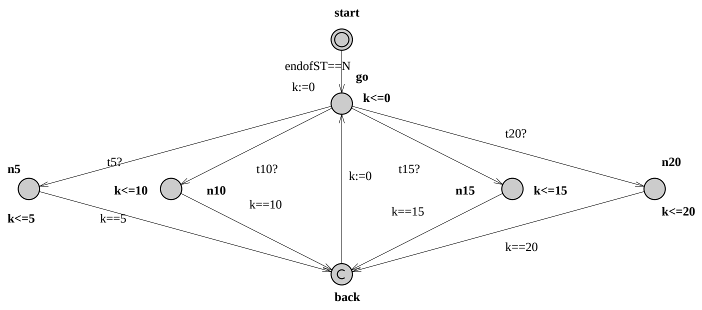

Fig. 6. The automaton *Timer*

Transitions normally don't take time in UPPAAL, but this does happen in the lift system. Each main loop consumes 1 millisecond. After each main loop, the station waits 0.5 millisecond to get messages from the bus. During the internal loop, the receiving and sending messages take 1 millisecond. Before sending a SYNC message, stations delay 1.5 milliseconds. Before sending a state message, stations delay 2 milliseconds. This is all the timing information in the normal operation phase. We use *Timer* to express time consumption by transitions; this

idea is borrowed from [9]. The guard 'endofST==N' makes sure that the *Timer* is only used in normal operation, where N is the number of lifts in the system. In node 'go', time is constrained to not progress at all. This means that in order for time to progress, one of the edges 'tn?' must be taken; where $n \in \{5, 10, 15, 20\}$ expresses the amount time of delay. These edges then lead to nodes where time can progress with the corresponding number of time units, where after control returns immediately to the 'go' node.
Concluding, the four problems in the original system are:

1. The relay between the first and second lift is re-opened at the wrong moment;
2. The SETREF buttons at two lifts are pressed in the start-up phase;
3. An UP or DOWN button is pressed and released at an inappropriate moment;
4. Two UP or DOWN buttons at different lifts are pressed at the same time, and one is released at an inappropriate moment.

4 Analysis of the redesign

Since the redesign does not change the desired external behavior of lifts, the UPPAAL model of the redesign should satisfy all the requirements in Section 2.2. We formulate those requirements in the UPPAAL requirement specification language, and verify them, sometimes with the help of test automata, to check whether the redesign solves problems 3 and 4. We do not give the definition and explanation of the UPPAAL requirement specification language [11]; we expect that the formulas in this section can be understood without difficulties.

4.1 Expressing the requirements

We first check deadlock freeness. This can be translated into the UPPAAL requirement specification language directly:

– A[] not deadlock

The redesign satisfies this property, which indicates that the solution from the developers solves the first problem found in [7]. In the implementation of the lift system, the delay for each STARTUP message is 24 milliseconds. In the UPPAAL model, a delay of 6 milliseconds for each STARTUP message is already enough to solve this problem.

Liveness I says that buttons on a lift can be pressed and released whenever the user wants, and that the system will respond to this. After implementing the main loop in the UPPAAL model, it is always possible to press or release buttons. So for the redesign, Liveness I becomes trivial.

Liveness II says that if an UP or DOWN button is pressed and not released and no other button is pressed, all lifts will move. In the UPPAAL requirement specification language, it is impossible to express this property. Fortunately, according to [1], we can transform this property into a test automaton, in which an approach is developed to model-checking of timed automata via reachability testing. The idea is to create a 'bad' state in the test automaton and let the verifier check whether the system can reach this state. If it does, the system violates a certain property.

The test automaton may need some extra 'decorations' for the verification purpose. In principle, with the test automaton we can express all scenarios we want to check. As this would lead to a possibly infinite state space, some scenarios which are not interesting can be abstracted away. For example, in the lift system, the buttons can be pressed and released many times. We consider only those scenarios where a button on one lift is pressed and released at most once. The automaton for the Liveness II requirement is depicted in Fig. 7.

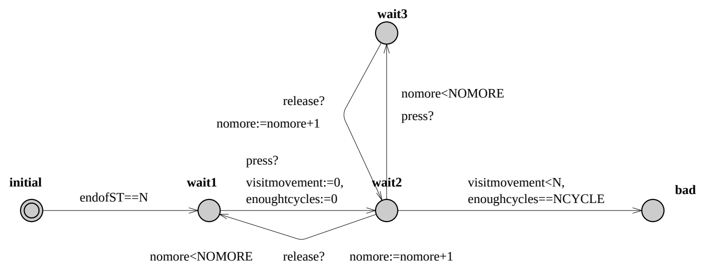

Fig. 7. The test automaton for Liveness II

We add new synchronizations between the *Interface* automata and the test automaton via *press* and *release* channels, to model the number of pressing and releasing actions. In the test automaton only one pressing and releasing per lift can take place. *nomore* is a variable that is used to block more pressing and releasing actions. This test automaton is used to express that if a button is pressed and not released any more, after some period of time (modeled by variable *enoughcycles*) all the lifts will move. We now check whether the test automaton can reach the node 'bad'. If the test automaton reaches the node 'bad', it means that not all the lifts have moved and the system violates property Liveness II.

– A[] not testautomaton.bad

Test automata are also used to model and check the other two safety properties.

With Liveness II, we could check that if one button is pressed, all the lifts reach their 'activemovement' or 'passivemovement' node within a certain amount of time. What we do not check is whether they move in the same direction. Safety I demands that whenever a lift moves, all the other lifts move simultaneously in the same direction. The corresponding test automaton is depicted in Fig. 8. This test automaton waits for one lift to reach the 'activemovement' node, which is detected by a synchronization on channel 'go?' between *Station* and this test automaton. The test automaton then checks whether the other lifts move in the

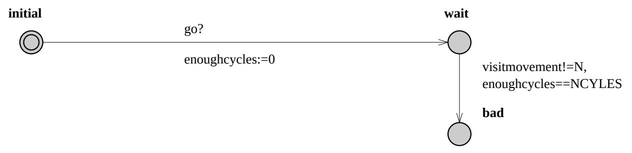

Fig. 8. The test automaton for Safety I

same direction (modeled by guard ‘visitmovement!=N’) within a certain amount of time (modeled by ‘enoughcycles==NCYCLES’).

Safety II states that there will be no movement when no button has been pressed. The corresponding test automaton is depicted in Fig. 9. The variable

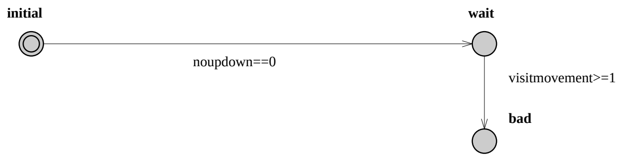

Fig. 9. The test automaton for Safety II

noupdown (meaning no UP or DOWN button pressed) is used to block all pressings of buttons in the *Interface*s. Now we can check whether it is still possible for the lifts to reach movement nodes (modeled by ‘visitmovement≥0’).

The redesign satisfies requirement Safety II, and violates requirements Liveness II and Safety I. We will discuss the diagnostic traces and the reasons in the next section.

4.2 Problems

The developers invented flags to solve the third problem found in [7]. These flags seem to solve the error scenario described in [7]. But during the testing phase, the developers encountered a new error; again the cause for this error was not clear to them. We have built a UPPAAL model (see Section 3) for the redesign and checked it. Liveness II turns out to be violated. We first investigate the diagnostic trace generated by the model checker in UPPAAL, and then give the reason why the solution from the developers fails. The generated diagnostic trace contains 256 transitions; we used the graphic simulation tool in UPPAAL to analyze it.

Initially all the flags are 0. When an UP button is pressed on one station (A), ECHO will be set and the state of station A will change to UP. Station A sends an UP message. The other stations will set the CAN flag and change their state to

UP. Suppose the button is released again. The flag of station A does not change, but its state will change to STANDBY (see the main loop in Fig. 5). Station A will send a STANDBY message which the others will adopt. When they have adopted this state, and if they receive another STANDBY message, the CAN flags of the other stations will be reset. After a short while all CAN flags in the network are 0, ECHO of station A is 1, and all the states of the stations are STANDBY. Suppose now that an UP button of another station (B) is pressed. Station B will send an UP message. Station A will receive this but cannot change its state because ECHO is set. When it is station A's turn to use the bus it will therefore send a STANDBY message. Station B will receive this STANDBY message, and it will not count enough UP messages. The whole counting procedure has to start over again. Station B will send an UP message. The other stations will adopt this state and send a UP message. But when it is station A's turn, again since ECHO is set, it will send a STANDBY message and station B will again not count enough UP messages. It is clear that the ECHO of station A should be reset to get out of this situation, but that can only happen when the state of the station is STANDBY, a STANDBY message is received, and it is this station's turn to use the bus. For station A this never happens. As a result, the whole system will never move, even when an UP button is pressed.

The test automaton detects this problem. Even though the solution of the developers has some virtue, they seem not to have taken into account that the main reason for the third problem lies in the fact that the active station immediately changes its state to STANDBY after a button is released. Their solution was directed to block state changes to the active station after its state has changed to STANDBY. This is not the heart of the problem and therefore the problem remains in the redesign.

The fourth problem found in [7] is also still in the redesign. The redesign violates Safety I property. The reason resembles what is already explained in [7]. This is not very surprising, since the fourth problem was unknown to the developers at the time of the redesign.

5 A new solution

In this section, we refine the solution proposed in [7] in such a way that it corresponds with UPPAAL and resemble to the solution from the developers. The key point why our solution differs from the flags added into the redesign is that our solution creates a link between the state change of a station and the turn of the station to use the bus. This idea was already mentioned in the μCRL model [7], but it was not further specified. With the more exact model of the redesign, including the main loop, and using the idea of the flags the developers came up with, now we work out the idea in detail.

The new flags are called CHANGE and ACTIVE. They are assigned to each station. CAN and ECHO are no longer a part of the new solution. When ACTIVE is 1, the corresponding station is active; otherwise, the station is passive. CHANGE of a station is set when there is a button pressed or released at this station (through the main loop). This is used to remember that the ACTIVE flag at this station must change from active to passive, or vice versa. Only when the station gets its turn to use the bus, this change will actually happen. If one station wants to become active, it has to make sure that there are no other active stations in the system, by checking whether the state of the message from the bus is STANDBY. If the CHANGE of a station is set, this station does not change its state until it is its turn to use the bus to make a decision. CHANGE is reset together with a setting or resetting of ACTIVE.

Changing the new flags has no effect on the automata *Interface*, *Bus* and *Timer*. They are exactly the same as in the redesign. Only the automaton *Station* has undergone crucial changes. We will not explain the new *Station* automaton in detail, more information can be found in [10]. All requirements have been checked successfully on the model with this new solution. In particular, problem three and four are resolved.

6 Concluding remarks

In this paper, we have reported an industrial case study on applying formal techniques for the design and analysis of a distributed system for lifting trucks. Our work can be considered as one piece of evidence that formal verification techniques are mature enough to be applied in industrial projects.

The lift system has been analyzed in the process algebraic language μCRL [7]. Four problems were found. Three of them were also found by the developers during the testing phase. They proposed solutions and made a redesign. But they faced a new problem. The redesign was then modeled in UPPAAL. The analysis in Section 4 has produced some interesting results. The first is that the redesign does not satisfy all the requirements. Second, the redesign does not solve all the problems with the original design. Only one problem is solved by adding time delays. The third problem, for which those flags were developed, and the fourth problem are not solved. Third, our solution in [7] was refined, and will be implemented in the new release of the lift system.

Since more details of the lift system are taken into account in the UPPAAL model, the state space of the redesign increases dramatically. In [7], we could analyze the μCRL model with up to five lifts. With the current version of the μCRL toolset, we can get up to seven lifts on a cluster at CWI, owing to a distributed state space generation algorithm. For the UPPAAL model of the redesign, we could only manage the analysis for systems with three lifts. The requirements were checked on a 1.4 GHz AMD Althlon$^{\text{TM}}$ Processor with 512 Mb memory.

Acknowledgments We thank the developers of the lift systems for their collaboration and fruitful discussions. We thank the anonymous referees for their useful comments.

References

1. L. Aceto, A. Burgueno and K.G. Larsen. Model checking via reachability testing for timed automata. In *Proceedings of 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1384, pp. 263-280. Springer-Verlag, 1998.
2. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and Y. Wang. Automated analysis of an audio control protocol using UPPAAL. *Journal of Logic and Algebraic Programming*, 52-53:163–181, 2002.
4. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.
5. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings of 13th Conference on Computer Aided Verification*, LNCS 2102, pp. 250–254. Springer, 2001.
6. Robert Bosch Gmbh, Postfach 30 02 40, D-70442 Stuttgart, Germany. *CAN Specification. Version 2.0*, 1991.
7. J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1/2):21–56, 2003.
8. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.
9. K. Havelund, K.G. Larsen and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In *Proceedings of 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, LNCS 1601, pp. 277-298. Springer-Verlag, 1999.
10. B. Karstens. *Formal verification of the redesign of a distributed lift system using UPPAAL.* Master thesis, Utrecht University, 2003. Available at `http://www.cwi.nl/~pangjun/research/liftredesign.ps`
11. K.G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
12. M. Lindahl, P. Pettersson, and Y. Wang. Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer*, 3(3):353–368, 2001.