Adapting the Uppaal Model of a Distributed Lift System

Wan Fokkink[1,2], Allard Kakebeen[3], and Jun Pang[4]

[1] Vrije Universiteit, Theoretical Computer Science, Amsterdam, The Netherlands
[2] CWI, Embedded Systems Group, Amsterdam, The Netherlands
[3] INRIA Sophia Antipolis, Everest Project, Sophia Antipolis, France
[4] Universität Oldenburg, Safety-Critical Embedded Systems, Oldenburg, Germany
`wanf@cs.vu.nl` `Allard.Kakebeen@sophia.inria.fr`
`jun.pang@informatik.uni-oldenburg.de`

Abstract. In [6] an existing distributed lift system was analyzed using the process algebraic toolset μCRL, and in [11] the redesign of this system was analyzed using the timed automata based toolset Uppaal. We adapt and extend the Uppaal model. Firstly, we refine the synchronization mechanism between lifts, to explain a new problem that was reported by the developers of the lift system, and to propose a solution for it. Secondly, we allow a lift to enter a halt state, after which the entire system should make an emergency stop, for instance because a lift meets a maximum height threshold. Using the Uppaal model checker it is verified that the adapted lift system satisfies the system requirements.

1 Introduction

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 specification of such a protocol 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 theorem prover. This formal reasoning can detect errors and suggest ways in which the system can be improved or optimized.

To achieve more confidence regarding the verified system, the formal model can be refined by adding more details, flaws in the model (detected using e.g. model checking) can be repaired, and extensions of the functionality can be included in the model. In this paper, we report on some experience gained by adapting the Uppaal model from [11] 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, and each lift has its own micro-controller. This system is being 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, which are connected in a ring network, operate synchronously. It consists of an initialization phase, in which all lifts get a unique identity, and a normal operation phase. When in the latter phase say an UP button on a lift is pushed, this lift leads the synchronous upward movement of all lifts until its UP button is released again. Special situations, such as when UP buttons at different lifts are pushed at the same time, have to be taken into account.

In order to explain and repair some detected bugs in the lift system, it was initially specified in the process algebraic language μCRL, and analyzed by means of model checking using the μCRL toolset [3]. This work was reported in [5, 6]. In a redesign of the lift system, the developers experienced a new problem. Since this problem involved exact timing information, and details of the system that had been abstracted away in the μCRL model, a more detailed model was specified in UPPAAL [10]. Using the graphic simulation tool in UPPAAL, the reason for the problem in the redesign was explained, and a solution was proposed. Moreover, it was shown using model checking that the UPPAAL model with this new solution satisfied all requirements. The solution was incorporated in the latest release of the lift system in early 2004. This work was reported in [11].

At the end of 2004, the developers of the lift system involved us in two matters regarding the coming release. Firstly, the developers reported that a new bug could sometimes occur when an UP button was pressed and almost immediately released again. We refine the synchronization mechanism between lifts in the UPPAAL model from [11], to capture this new problem and propose a solution for it. Secondly, the developers wanted a more polished solution for the situation where the system has to make an emergency stop because, for instance, one of the lifts meets a maximum height threshold. This feature of the system had been abstracted away in the μCRL and the original UPPAAL model. In our new UPPAAL model, we allow a lift to enter a special "halt" state, which is spread to the other lifts, upon which they all halt. The main challenge is how to move from this halt state to a standby state, as this requires that the main authority shifts back from the lift that initiated to halt state to the lift that controlled the movement.

During the adaptation of the UPPAAL model, we made several initial design errors, which were detected in the model checking phase. In this paper we explain our ultimate solutions for the synchronization mechanism and the halt state, and report on some of the initial design errors. Moreover, during the model checking exercise we detected a flaw in the UPPAAL model from [11] (which does not occur in the real implementation of the lift system). This flaw in the model had gone unnoticed due to a too restrictive test automaton in that paper. We explain how this flaw in the model can be repaired. We have shown using the UPPAAL model checker that our solutions are correct, at least for ring networks of size three, and with respect to the scenarios in our test automata.

A first aim of the current paper is to add yet another experience report on the use of formal methods in industry. Our collaboration with the company that builds the lift system has continued over the last five years. This experience is quite unique, in the sense that formal methods and tools (μCRL and UPPAAL) have been applied to the original lift system and its redesigns in three subsequent case studies. Over the years, the team of developers remained the same, but the team from the formal methods side has changed at each case study. In Section 6 we will draw some conclusions on the use of formal methods in long term industrial development, on the basis of these case studies.

A second aim is to communicate our experiences with adapting an UPPAAL model. Also it is explained how we used the UPPAAL model checker, with the help of test automata and decoration variables.

The developers acknowledge the usefulness of formal verification for their redesign. The new synchronization mechanism was included in the latest release of the lift system. Our specification of the special halt state will become part of the next release. The developers are now more confident in the correct functioning of the redesigned lift system. They stress that applying formal methods in the early design phases would save them testing effort and cost.

The paper is structured as follows. In Section 2, we provide an informal high-level description of the lift system, together with an explanation of our UPPAAL model of this system. Section 3 presents the system requirements that we want to verify. In Section 4 we present the refined synchronization mechanism between lifts, and explain in detail how we model checked the resulting UPPAAL model. In Section 5 we present the extension with a special halt state for emergency situations, and again describe the model checking exercise. Section 6 contains the conclusions. And finally Section 7 contains the most important automata of the UPPAAL model of the lift system.

2 Uppaal Model of the Lift System

2.1 High-level system description

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: SETREF, UP and DOWN. Pressing a SETREF button on a lift is the only way a run of the system can start. If an UP or DOWN button on a certain lift is pressed, all lifts in the system are meant to move up or down at the same time. If the UP button at a lift has been pressed, the DOWN button at this same lift cannot be pressed before the UP button is released.

Movement of the lift system is controlled by means of a micro-controller. Each lift has its own micro-controller, called station here. Stations can adopt four different states: STARTUP, STANDBY, UP and DOWN. The state of a station 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 contains two pieces of information: the position of the sender station and the type of the message. There are two types of messages: state messages and sync messages. State messages broadcast the state of the sending station to the other stations, while sync messages initiate physical movement. In response to a sync message, the receiving station transfers its state to the motor of the lift, which causes movement. If the station is in the state 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 [4]. The CAN bus is a 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 claim 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 lift system can start to operate, there is a start-up phase has been designed in which each station finds out its position in the network and the total number of lifts in the network. This start-up phase is part of our Uppaal model, but we abstract away from it here, as it is identical to the specification of the start-up phase in the original Uppaal model. See [11] for a detailed description of the start-up phase.

When the start-up phase has finished, each station has been assigned a unique position and is in the state STANDBY. Then the normal operation phase starts, which is described in some detail in the remainder of this section. During normal operation, stations claim the bus in the same order cycle after cycle. A station knows whether it is its turn to claim the bus by checking the position of the sender station in the last received message. The state of a station changes from STANDBY to UP or DOWN when its UP or DOWN button is pressed, respectively. A station where this happens is called an active station. The active station sends an up or down message, according to the button that was pressed at the station. Each passive station changes its state according to the messages it receives, and when it is its turn to claim the bus it broadcasts its state. These state messages are received by all other stations, and the ordered sending of messages makes sure that the active station counts no more than one message from each station. When the active station counts enough up (or down) messages, it concludes that all lifts are ready to move. Then the active station broadcasts a sync message, after which all lifts move. In contrast to passive stations, the state of the active station can only change when the pressed button is released again. In that case its state changes to STANDBY and the station becomes passive again.

2.2 Uppaal model

Uppaal [10] is a toolset for verifying timed systems, which are modeled as networks of timed automata [2], extended with global shared variables. Clock variables can be associated to a transition or a node. In a transition, clock variables can be reset or used as a guard. There are a graphical editor for system specification, a simulator and a model checker. During the design phase, the simulator is used 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 techniques. Symbolic techniques are used to reduce the verification of modal logic formulas to solving simple reachability constraints.

The UPPAAL (version 3.4.11) model of the lift system consists of four automata: *Bus*, *Timer*, *Station* and *Interface*. The automaton *Bus* models the CAN bus, and the automaton *Timer* models time delays. For each lift in the system we create two automata: *Station* and *Interface*, where *Station* models the microcontroller, while *Interface* captures the pressing and releasing of buttons on the lift. These last two automata can be found at the end of this paper, in Section 7. The complete UPPAAL model is available at `www.cs.vu.nl/~wanf/haltmodel3.xml`. Here we only provide sufficient explanations to present our adaptations of the original UPPAAL model and the analysis of this adapted model. A more detailed explanation and motivation can be found in [9].

Fast and main loop Each station performs two different loops. In the so-called *fast loop*, a station can get a message from the bus, and when it is a station's turn to claim the bus, it sends a message to the bus intended for the other stations. Furthermore, the active station can count state messages and initiate movement of the whole system, by means of a sync message. In a *main loop*, a station synchronizes with its interface, to obtain information about which button on the lift (if any) has been pressed or released. Such a main loop takes place after a fixed number of cycles from the fast loop.

The two loops were implemented separately because communication with the bus is relatively fast. The separation leads to faster communication between the lifts, which is essential for the safe functioning of the system, as else the response time of the system would become too slow.

Flags In [11], two flags CHANGE and ACTIVE were introduced in the automaton *Station*, as an improvement over two flags in the implementation of the lift system. The developers of the lift system acknowledged that this improvement solved a detected bug in the system, and included the new flags in its redesign.

When ACTIVE is set, the corresponding station is active; otherwise, the station is passive. CHANGE of a station is set when at this station a button is pressed or released; this update is communicated to the station through the main loop. The CHANGE flag is used to remember that the ACTIVE flag at this station must change from passive to active, or vice versa. CHANGE is reset together with a setting or resetting of ACTIVE (or if in the meantime a button is released or pressed again). This first change happens as soon as the station gets its turn to claim the bus, and the incoming message carries the state STANDBY.

Bus We omit a precise description of the internals of the automaton *Bus*, and view it as a black box that regulates the distribution of messages in the fast loop. In the Uppaal model, there are two channels for communication between the bus and the stations, and global shared variables are used for data transfer over these channels. When a station wants to send a message to the bus, it has to instantiate values for some global variables, for instance the sender's identity and state. When communication takes place, the values of those variables are saved to local variables of the bus. In a similar fashion, messages are sent from the bus to the stations.

Timer Transitions normally do not take time in Uppaal, but they do 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 fast loop, the receiving and sending messages take 1 millisecond. Before sending a sync message, stations delay 1.5 millisecond. And before sending a state message, stations delay 2 milliseconds. The automaton *Timer* expresses this time consumption by means of transitions; this idea is borrowed from [7].

3 Requirements

The desired behavior of the lift system is captured in five requirements it has to fulfill, taken from [11]. These requirements were formulated together with the developers of the system.

1. *Deadlock freeness*: The system never ends up in a state where it cannot perform any action.
2. *Liveness I*: If all buttons are released, the system will eventually get to a state in which all lifts are standby.
3. *Liveness II*: If exactly one UP (or DOWN) button is pressed and not released, then all lifts will eventually move up (or down).
4. *Safety I*: If one of the lifts moves, then all other lifts move simultaneously (that is, within one cycle of the fast loop) in the same direction.
5. *Safety II*: If the lifts move, then an appropriate button was pressed.

The model checker of Uppaal allows to check formulas over a rather weak temporal logic. In particular, *Liveness II* and *Safety I* cannot be expressed in this logic. In [1] an approach was developed for model checking such properties via reachability testing. The idea is to transform the property into a so-called *test automaton*, which is placed in parallel to the Uppaal model of the system. Such a test automaton is typically built from a specific scenario (e.g., a fixed sequence of button presses and releases), and contains a 'bad' state which can only be reached if the corresponding property is violated.

The test automaton may need some extra information that is not being maintained in the Uppaal model of the system (such as how often a certain loop has been taken). This information can be added to the model, without influencing its functional behavior, in the form of so-called *decoration variables.*

In our verifications of two versions of the Uppaal model of the lift system, which will be described in Sections 4 and 5, we made extensive use of test automata and decoration variables. We performed model checking with respect to networks of two or three lifts.

In the test automata, station 1 will play the role of active station. That is, the UP button at station 1 is the first button to be pressed, making station 1 active. We note that this is not a real limitation, in the sense that the network is fully symmetric (i.e., all stations exhibit the same behavior).

4 Sync Flag

The developers of the lift system informed us that a deadlock had occurred. After some testing at their premises, empirical evidence showed that it may occur if an UP (or DOWN) button is released shortly after it was pressed. Since this deadlock was not detected using the UPPAAL model from [11], it appeared that a crucial aspect of the system was missing in that model. The developers were of the opinion that the bug was most likely in the synchronization mechanism between the stations.

Discussions with the developers brought to light the fact that synchronization of the stations is implemented in a somewhat different fashion than as was specified in the UPPAAL model. In the real system there is an extra SYNC flag at each station, which is missing in the UPPAAL model.

When the active station counts enough up (or down) messages, in the UPPAAL model, this station initiates movement straight away by sending a sync message to the other stations. But in the real system, at each station there is an extra SYNC flag, which is set if the state of the station is UP (or DOWN) and there is no obstruction to send output to the motor. The active station only sends a sync message when the SYNC flag is set at each station. When output is sent to the motor, the SYNC flag at that station is reset. The SYNC flag guarantees that output ports of the stations to their motors are in sink. Else it might be the case that one station moves while another does not, for instance because the latter reached its highest position.

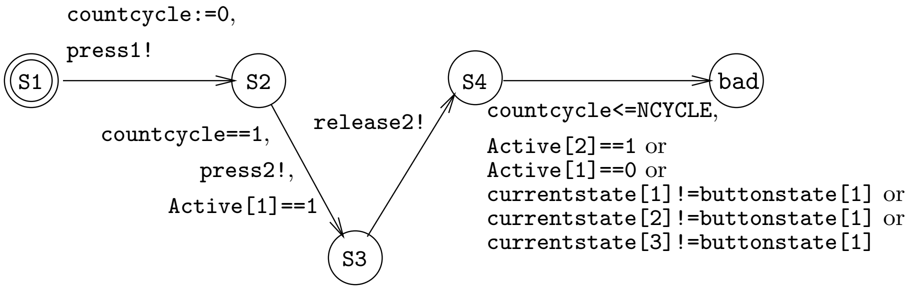
At first sight, it makes sense to abstract away from the SYNC flag in the UPPAAL model, as it does not take into account obstructions to the output ports. However, the SYNC flag can have an influence on the functional behavior, even in the absence of such obstructions. We therefore adapted the UPPAAL model from [11] to include the SYNC flag, and analyzed by means of the UPPAAL model checker whether the adapted model satisfies the requirements from Section 3.

Deadlock freeness can be expressed in the modal logic of UPPAAL:

```
A[] not deadlock
```

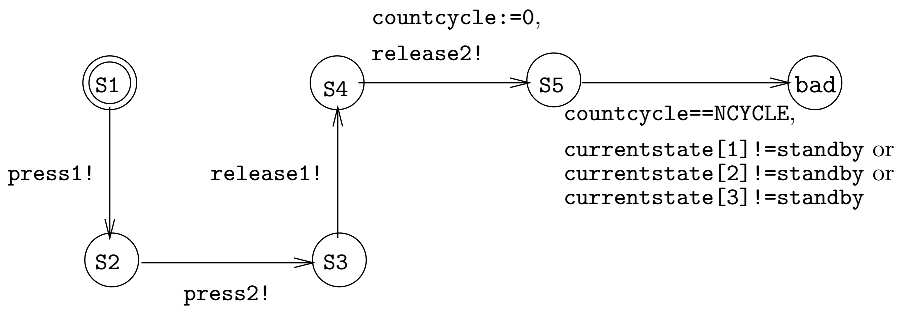
where `deadlock` is a predefined predicate in UPPAAL that holds for all deadlock states. We checked this property with respect to a number of scenarios (i.e., test automata). Initially this property was violated. Analysis of the error trace showed that, in line with reports from the developers, the deadlock may occur if an UP (or DOWN) button is released shortly after it has been pressed. Namely, as said before, the SYNC flag is reset when output is sent to the motor. But if the UP button is released shortly after it has been pressed, it may be the case that a SYNC flag at some station is set, but never reset, because no output is sent to the motor. The simple solution for this problem is to reset SYNC flags also when a button is released. We included this solution in our UPPAAL model, upon which no further deadlocks were detected.

Parallel button presses do not have an effect. That is, suppose that a button at some station is pressed. If (before the release of this button) a button at another lift is pressed and released, then this does not effect the states of the stations. We formulated this in a test automaton, in which initially the UP button at station 1 is pressed, expressed by the flag `press1!`. This button press makes station 1 active; the guard `Active[1]==1` makes sure that this happens before the UP button at station 2 is pressed (`press2!`), as else station 2 might get active instead of station 1. Finally the button at station 2 is released (`release2!`). The `bad` state can be reached if as a result the state of one of the stations is not in sink with the button state of the active station 1. The test automaton uses a decoration variables `countcycle`, which in the model is increased by one at every fast loop, together with a parameter `NCYCLE` (which we instantiated with 6 for two lifts, and with 13 for three lifts). The guard `countcycle<=NCYCLE` on the last transition guarantees that the scenario covers only a bounded number of fast loops, as otherwise the property could not be model checked. Without the guard `countcycle==1` on the second transition the `bad` state could be reached, as it requires one cycle of the fast loop before all stations have attained the same state as `buttonstate[1]`.



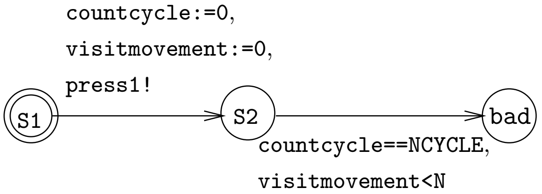
A model checking exercise with respect to our model in parallel to the test automaton above (for three lifts) showed that the `bad` state in the test automaton cannot be reached. In view of this positive model checking result, in the test automata to follow regarding the liveness and safety requirements, we do not take into account parallel button presses.

Liveness I was checked for a number of scenarios. Below a test automaton is presented in which first the UP button at station 1 is pressed (making it active), next the UP button at station 2 is pressed, then the button at station 1 is released (making station 2 active), and finally the button at station 2 is released. As before, `countcycle` and `NCYCLE` are used to make the scenario bounded. The `bad` state can be reached if after `NCYCLE` fast loops the stations are not all standby. (Again we instantiated `NCYCLE` with 6 for two lifts, and with 13 for three lifts).



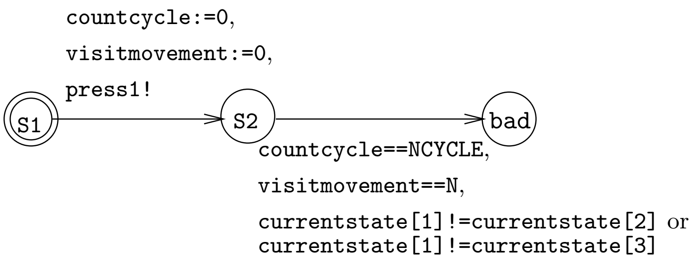
A model checking exercise with respect to our model in parallel to the test automaton above (for three lifts) showed that the `bad` state in the test automaton cannot be reached.

Liveness II was verified using a test automaton in which the UP button at station 1 is pressed and not released; *Liveness II* requires that eventually all lifts will start moving. As before, `countcycle` and `NCYCLE` are used to make the scenario bounded. In the model, the decoration variable `visitmovement` is increased by one every time a lift starts moving. Furthermore, `N` denotes the number of lifts in the system. If the UP button at station 1 is pressed and not released, and at the deadline (`countcycle==NCYCLE`) not all lifts have started moving (`visitmovement<N`), then the `bad` state is reached.



A model checking exercise with respect to our model in parallel to the test automaton above showed that the `bad` state in the test automaton cannot be reached.

The test automaton that was used in [11] for checking *Safety I* captures only a quite restricted collection of scenarios. We constructed the following more elaborate test automaton. It has a similar structure as the previous test automaton. Suppose that the UP button at station 1 is pressed. The `bad` state can only be reached if ultimately (`countcycle==NCYCLE`) all lifts move (`visitmovement==N`) while not all stations are in the same state (`currentstate[1]!=currentstate[2]` or `currentstate[1]!=currentstate[3]`).

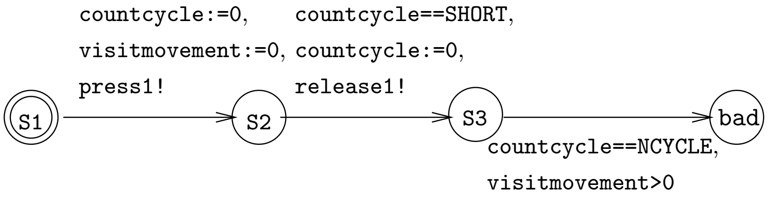


To our surprise, a model checking exercise with respect to our model in parallel to the test automaton above showed that *Safety I* was violated. We also checked this test automaton against the UPPAAL model from [11], and there too it was violated. In the model, lifts could actually move in opposite directions! This bug did not occur in a network with two lifts, but it did in a network with three lifts. Analysis of the error trace showed the reason for the bug: in the UPPAAL model (unlike the implementation), a global variable in the CAN bus maintains the message state; each station can read this variable. With three lifts, it is possible that a station receives a state message without processing it yet. Then another station may send a message to the bus, overwriting the message state variable before the first station reads it. The solution for this problem is simply to conform to the implementation, by introducing a message state variable at each station. We included this solution in our UPPAAL model, upon which *Safety I* was satisfied.

Finally, we verified *Safety II* in the same fashion as in [11]. The idea is to put a 'false' guard on all transitions in the *Interface* automaton that represent a button being pressed, and to add a flag `move` to the node capturing movement in the *Station* automaton, which is set if this node is visited. Now *Safety II* can be verified using the following modal formula:

```
A[] move == 0.
```

We also verified *Safety II* with respect to a number of scenarios in which a button is pressed and then released within a short time interval, expressed by the parameter `SHORT`. We verified that in such scenarios no lift moves. In the test automaton below, `SHORT` was given the value 3 (in case of three lifts).



5 HALT State

In the implementation of the lift system, the situation is taken into account where the system has to make an emergency stop, for instance because one of the lifts meets a minimum or maximum height threshold. This feature of the system was abstracted away in the UPPAAL model in [11]. The developers of the lift system asked us to propose a more polished solution for emergency stops, because in their implementation emergency stops were dealt with in a rather ad hoc fashion.

We extended the *Station* automaton, by allowing it to enter a special HALT state, which is spread to the other lifts, upon which they all halt. Adapting the model can be split into three tasks: (1) achieve HALT in the detecting station, (2) communicate this HALT state to the other stations, and (3) leave the HALT state to continue normal operation from the STANDBY state.

Detecting HALT In the *Station* automaton, we added a transition which allows a station (nondeterministically) to detect a halt notification, after which it changes its state to HALT. Initially, we allowed this transition to be taken only when the lift is in movement. However, according to the developers, in real life detection may also happen when a button was pressed but no movement has taken place yet. Therefore the latter was added as an extra possibility.

Spreading HALT The HALT state is spread to the other stations via the fast loop. When a station in state HALT gets its turn to claim the bus, it sends out a halt message, which makes the other stations take on the HALT state too.

Leaving HALT The hardest part is leaving the HALT state once the button that initiated the last movement has been released. The active station, at which this button was pressed, then needs to return to the STANDBY state to resume normal operation. It must therefore ignore further incoming halt messages from the other stations. So if a station is in HALT state, and its ACTIVE and CHANGE flags are both set (meaning that it is the active station and the button has been released), it adopts the STANDBY state, and spreads this state to the other stations via the fast loop.

We analyzed by means of the UPPAAL model checker whether the adapted model, including the HALT state, satisfies the requirements from Section 3. Two requirements need the proviso that HALT is not detected.

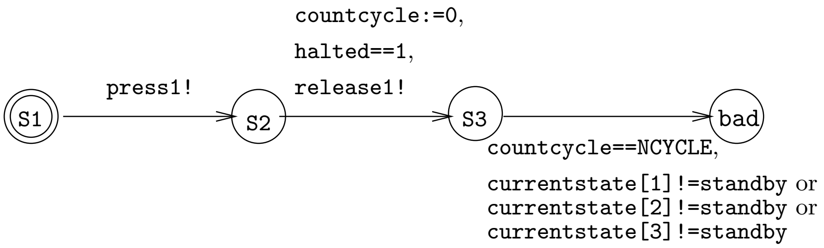
1. *Liveness II*: If exactly one UP (or DOWN) button is pressed and not released, *and* HALT *is not detected*, then all lifts will eventually move up (or down).
2. *Safety I*: If one of the lifts moves, *and* HALT *is not detected*, then all other lifts simultaneously move in the same direction.

Furthermore, together with the developers of the system we formulated one extra liveness requirement and one extra safety requirement.

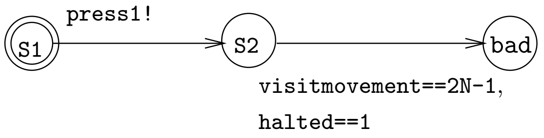
1. *Liveness III*: After HALT is detected, it is always possible for the system to get to a state where all stations are STANDBY.
2. *Safety III*: If HALT is detected, then within a certain amount of time a state is reached where no lift moves.

Deadlock freeness, *Liveness I* and *Safety II* could be verified as in the previous section. For the other requirements, a global decoration variable `halted` was introduced, to signal the detection of HALT. In the test automata for *Liveness II* and *Safety I*, we added a guard to ensure that the `bad` node can only be reached if `halted` is not set. With these adapted test automata, *Liveness II* and *Safety I* could be verified without problem.

Liveness III was checked against a test automaton that is similar to the test automaton that we used for *Liveness I* in the previous section. The main difference is that a guard `halted==1` was added, to express that HALT has been detected. A model checking exercise with respect to our model in parallel to the resulting test automaton showed that the `bad` state cannot be reached.



For *Safety III*, initially we required that after a detection of HALT, all lifts would stop moving within one cycle of the fast loop. However, this turned out to be too strict. If the lifts are moving, and the station detecting HALT has just sent a state message to the bus, it may be that two cycles of the fast loop are needed before all lifts have halted. *Safety III* with "four cycles of the fast loop" substituted for "a certain amount of time" does hold (in case of three lifts). That is, all stations except the one that detects HALT move at most twice after this detection; so in general there can be at most `2N-2` movements from the moment HALT is detected. In the model, the value of the decoration variable `visitmovement` (which is increased by one at every movement of a lift) is set to zero as soon as `halted` is set. The `bad` state is reached if `visitmovement>=2N-1` and `halted==1`. We successfully checked the following test automaton against our model.



6 Concluding Remarks

In this paper, we have reported on an industrial case study in which formal techniques were applied for the analysis of a distributed system for lifting trucks. Our work can be considered as one more piece of evidence that formal verification techniques are sufficiently mature to be applied in the design of industrial systems. In particular embedded controllers appear to be well-suited for formal modeling and verification with model checking, as they tend to combine a high degree of complexity with a manageable state space.

A formal model is always an abstraction of the real system. The good thing is that this enables to study the core of a system, without superfluous details that may needlessly obscure the picture and increase the state space. A drawback however is that one may abstract away too much. In our case study, we saw two examples of this, regarding the UPPAAL model from [11]. Firstly, abstracting away from the SYNC flag meant that a bug in the implementation was missed. Secondly, using one global shared variable instead of different local shared variables at all stations induced a serious flaw in the model.

The latter flaw brings us to the use of test automata, as this flaw was initially missed due to a too restrictive test automaton. UPPAAL's modal logic is not very expressive. It is well-known that test automata can come to the rescue, to express different scenarios of a property that is outside the scope of the modal logic. However, this comes at a price. First of all, it means that only a subset of scenarios is verified. Furthermore, building a good test automaton can be quite laborious. Last but not least, a test automaton can itself be too restrictive or even flawed.

A strong point of formal models is that it is relatively easy to extend or adapt them, and then verify the adapted model. We experienced that it is very useful to have the ability to try different solutions to a problem (in this case the extra HALT state), and verify with model checking whether a solution is satisfactory. This was far easier than it would have been for the developers of the system to implement these different solutions and perform a substantial testing effort.

One has to keep in mind that an adaptation of the model can give rise to an adaptation of requirements, or to new requirements. Here we had to adapt *Liveness II* and *Safety I*, and introduced new requirements *Liveness III* and *Safety III*, for the extended model that includes a HALT state.

For us, the excellent graphical interface of UPPAAL has been invaluable, as it enabled the developers of the lift system to fully understand and comment on our formal models. We would like to emphasize the importance of establishing a good relationship between a formal methods group and a team of engineers. This relationship should be built on mutual trust and technical insight. Too often, a formal verification effort within industry is limited to a single case study. In general it would be much more fruitful to perform a series of case studies with the same group of engineers, and ideally with subsequent releases of the same system. This way the engineers get better acquainted with the formal methods approach, and the formal methods people get a better technical insight. Even more important, this way the results of a formal analysis can have a direct impact on the design of a system, and the strengths of formal models come to light. Namely, while developers may struggle to adapt the implementation and have to spend considerable testing effort, adaptation of the formal model and the subsequent model checking exercise tend to take relatively little effort.

7 Uppaal Automata of the Lift Model

We present the two most important automata of our Uppaal model. Start-up phase and the automata *Bus* and *Timer* are left out.

The automaton *Interface*, which is depicted in Figure 1, captures the buttons on a lift.

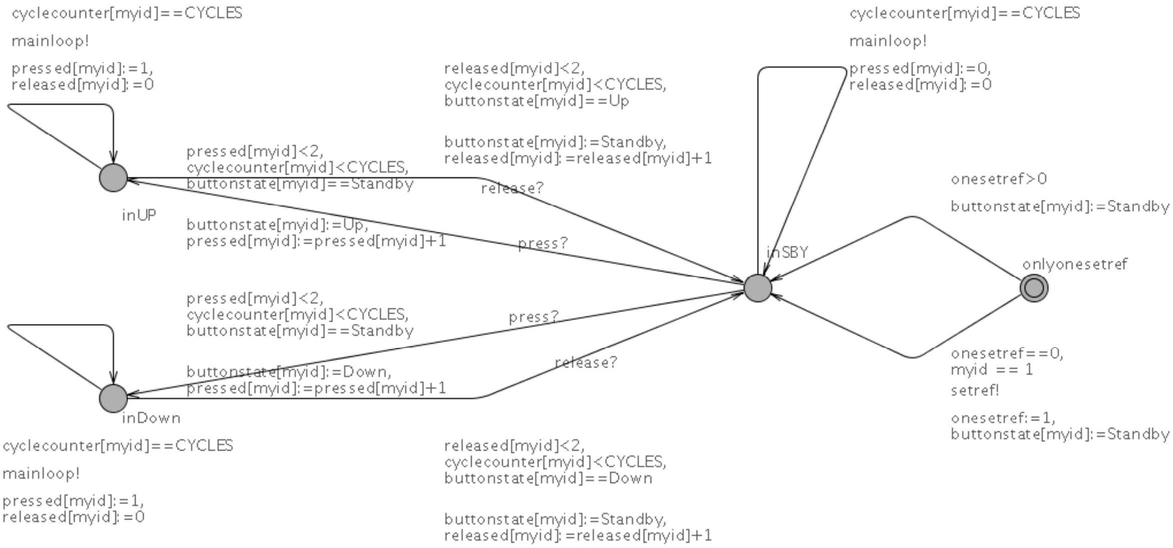

cyclecounter[myid]==CYCLES
mainloop!
pressed[myid]:=1,
released[myid]:=0
released[myid]<2,
cyclecounter[myid]<CYCLES,
buttonstate[myid]==Up
buttonstate[myid]:=Standby,
released[myid]:=released[myid]+1
cyclecounter[myid]==CYCLES
mainloop!
pressed[myid]:=0,
released[myid]:=0
pressed[myid]<2,
cyclecounter[myid]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

Fig. 1. The automaton *Interface*.

The automaton *Station* is depicted in two separate parts, which are joined together at the initial node `normaloperation`. At this node, two loops of a station can be performed: the main loop and the fast loop.

The main loop, which is depicted in Figure 2, 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 fast loops, which is modeled as a constant `CYCLES` in the Uppaal model. A counter `cyclecounter` is used to record the number of fast 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 the 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*.

In the fast loop, which is depicted in Figure 3, a station can do several things. First a station can get messages from the bus. Second, a station can send a

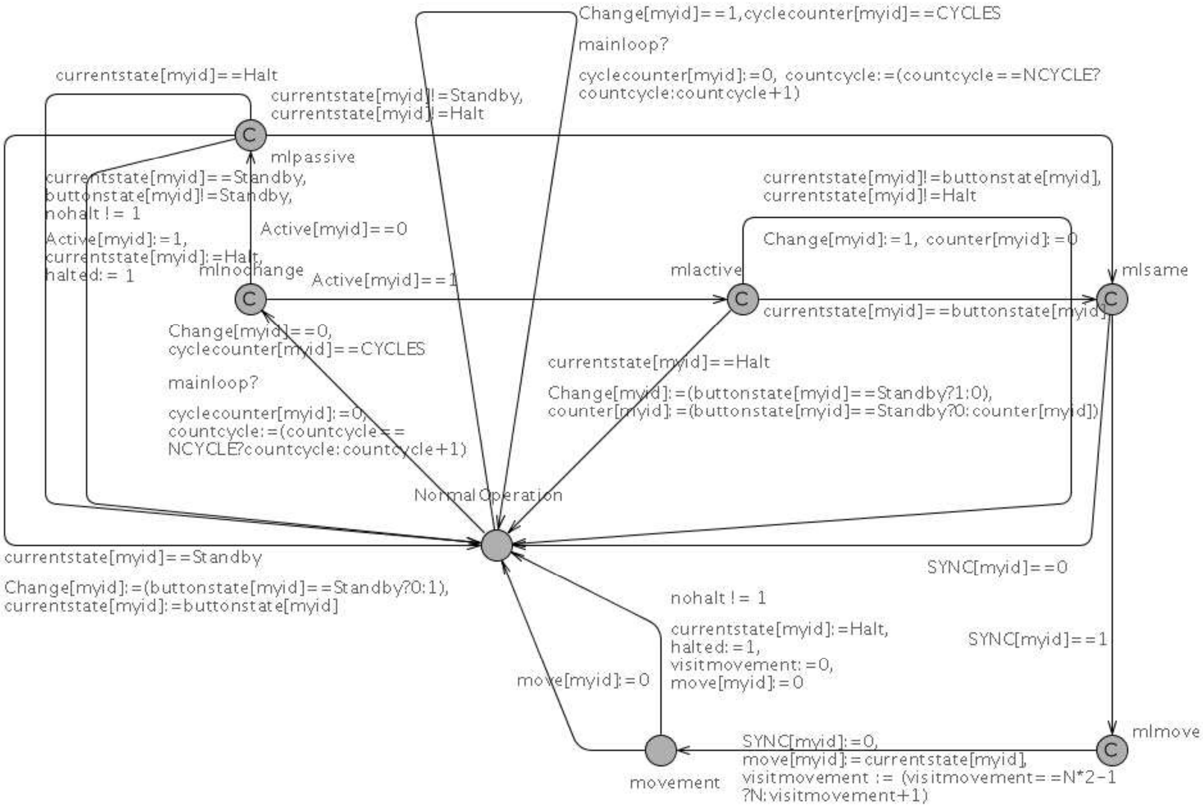

Change[myid]==1,cyclecounter[myid]==CYCLES
mainloop?
cyclecounter[myid]:=0, countcycle:=(countcycle==NCYCLE?
countcycle:countcycle+1)
currentstate[myid]==Halt
currentstate[myid]!=Standby,
currentstate[myid]!=Halt
mlpassive
currentstate[myid]==Standby,
buttonstate[myid]!=Standby,
nohalt != 1
Active[myid]:=1,
currentstate[myid]:=Halt,
halted:= 1
Active[myid]==0
mlnochange
Active[myid]==1
mlactive
currentstate[myid]!=buttonstate[myid],
currentstate[myid]!=Halt
Change[myid]:=1, counter[myid]:=0
mlsame
currentstate[myid]==buttonstate[myid]
Change[myid]==0,
cyclecounter[myid]==CYCLES
mainloop?
cyclecounter[myid]:=0,
countcycle:=(countcycle==
NCYCLE?countcycle:countcycle+1)
currentstate[myid]==Halt
Change[myid]:=(buttonstate[myid]==Standby?1:0),
counter[myid]:=(buttonstate[myid]==Standby?0:counter[myid])
NormalOperation
currentstate[myid]==Standby
Change[myid]:=(buttonstate[myid]==Standby?0:1),
currentstate[myid]:=buttonstate[myid]
SYNC[myid]==0
nohalt != 1
currentstate[myid]:=Halt,
halted:=1,
visitmovement:=0,
move[myid]:=0
SYNC[myid]==1
move[myid]:=0
mlmove
SYNC[myid]:=0,
move[myid]:=currentstate[myid],
visitmovement := (visitmovement==N*2-1
?N:visitmovement+1)
movement

Fig. 2. Part of the automaton *Station*: Main loop.

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`.

Acknowledgments We thank the developers of the lift system for their collaboration and fruitful discussions. Henk Barendregt and Frits Vaandrager provided useful feedback.

References

1. L. Aceto, A. Burgueno and K.G. Larsen. Model checking via reachability testing for timed automata. In *Proc. 4th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, 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. 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 *Proc. 13th Conference on Computer Aided Verification (CAV'01)*, LNCS 2102, pp. 250–254. Springer, 2001.
4. Robert Bosch Gmbh, Postfach 30 02 40, D-70442 Stuttgart, Germany. *CAN Specification. Version 2.0*, 1991.
5. J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: Analyzing a distributed lift system. In *Proc. 6th Workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, pp. 1–12, 2001.
6. 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.
7. K. Havelund, K.G. Larsen and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In *Proc. 5th AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99)*, LNCS 1601, pp. 277-298. Springer-Verlag, 1999.
8. B. Karstens. *Formal Verification of the Redesign of a Distributed Lift System using* UPPAAL. MSc thesis, Utrecht University, June 2003. Available at `www.phil.uu.nl/preprints/scripties/list.html`.
9. A. Kakebeen. *Extension and Formal Verification of a Distributed Lift System in* UPPAAL. MSc thesis, Radbout Universiteit Nijmegen, August 2005. Available at `www.cs.vu.nl/~wanf/kakebeen.doc`.
10. K.G. Larsen, P. Pettersson, and Y. Wang. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
11. J. Pang, B. Karstens, and W.J. Fokkink. Analyzing the redesign of a distributed lift system in UPPAAL. In *Proc. 5th Conference on Formal Engineering Methods (ICFEM'03)*, LNCS 2885, pp. 504–522. Springer-Verlag, 2003.

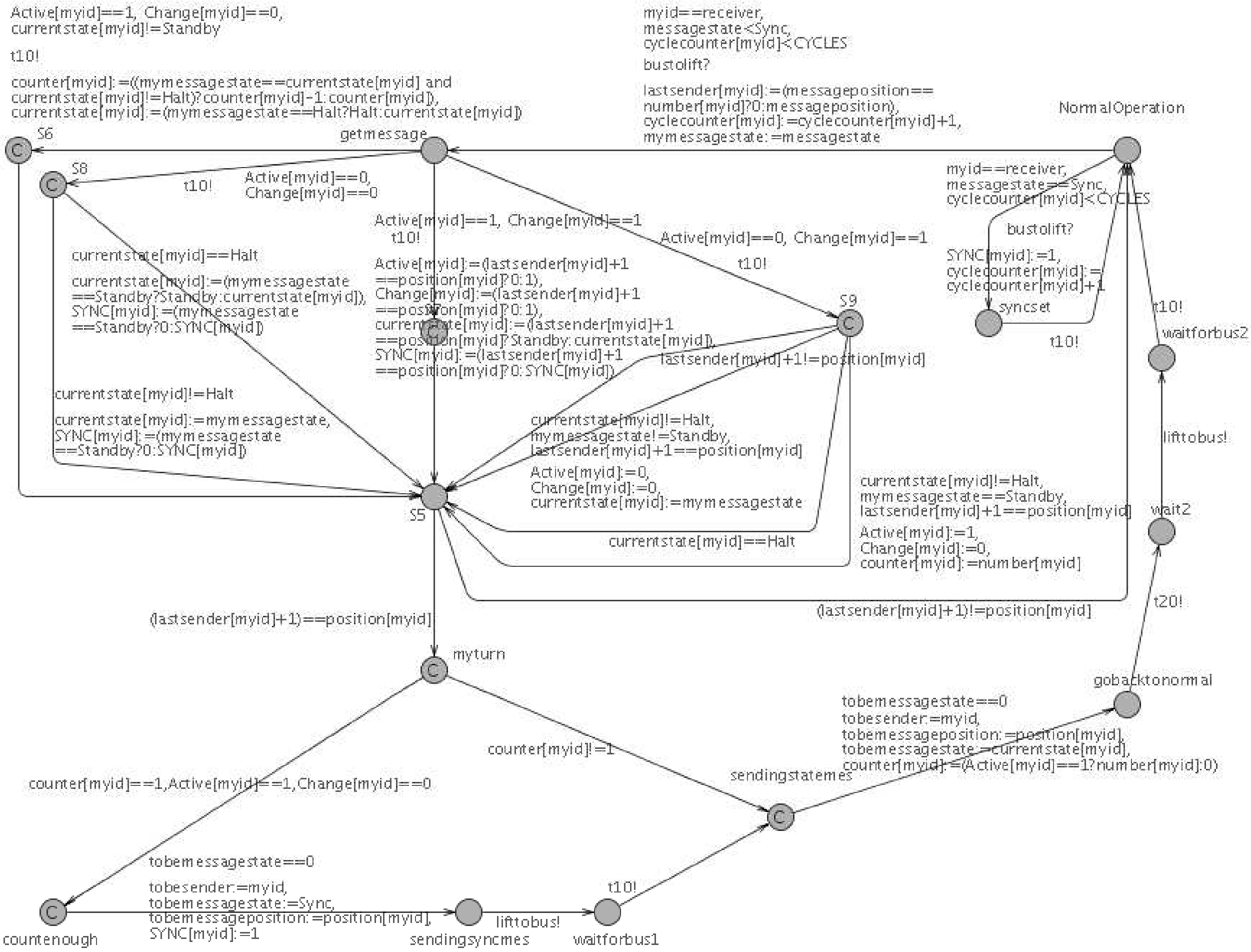

Active[myid]==1, Change[myid]==0,
currentstate[myid]!=Standby
t10!
counter[myid]:=((mymessagestate==currentstate[myid] and
currentstate[myid]!=Halt)?counter[myid]-1:counter[myid]),
currentstate[myid]:=(mymessagestate==Halt?Halt:currentstate[myid])
myid==receiver,
messagestate<Sync,
cyclecounter[myid]<CYCLES
bustolift?
lastsender[myid]:=(messageposition==
number[myid]?0:messageposition),
cyclecounter[myid]:=cyclecounter[myid]+1,
mymessagestate:=messagestate
NormalOperation
S6
getmessage
S8
t10!
Active[myid]==0,
Change[myid]==0
myid==receiver,
messagestate==Sync,
cyclecounter[myid]<CYCLES
bustolift?
SYNC[myid]:=1,
cyclecounter[myid]:=
cyclecounter[myid]+1
Active[myid]==1, Change[myid]==1
t10!
Active[myid]==0, Change[myid]==1
t10!
currentstate[myid]==Halt
currentstate[myid]:=(mymessagestate
==Standby?Standby:currentstate[myid]),
SYNC[myid]:=(mymessagestate
==Standby?0:SYNC[myid])
Active[myid]:=(lastsender[myid]+1
==position[myid]?0:1),
Change[myid]:=(lastsender[myid]+1
==position[myid]?0:1),
currentstate[myid]:=(lastsender[myid]+1
==position[myid]?Standby:currentstate[myid]),
SYNC[myid]:=(lastsender[myid]+1
==position[myid]?0:SYNC[myid])
S9
syncset
t10!
t10!
waitforbus2
lastsender[myid]+1!=position[myid]
currentstate[myid]!=Halt
currentstate[myid]:=mymessagestate,
SYNC[myid]:=(mymessagestate
==Standby?0:SYNC[myid])
currentstate[myid]!=Halt,
mymessagestate!=Standby,
lastsender[myid]+1==position[myid]
Active[myid]:=0,
Change[myid]:=0,
currentstate[myid]:=mymessagestate
lifttobus!
currentstate[myid]!=Halt,
mymessagestate==Standby,
lastsender[myid]+1==position[myid]
Active[myid]:=1,
Change[myid]:=0,
counter[myid]:=number[myid]
wait2
S5
currentstate[myid]==Halt
(lastsender[myid]+1)!=position[myid]
t20!
(lastsender[myid]+1)==position[myid]
myturn
gobacktonormal
tobemessagestate==0
tobesender:=myid,
tobemessageposition:=position[myid],
tobemessagestate:=currentstate[myid],
counter[myid]:=(Active[myid]==1?number[myid]:0)
counter[myid]!=1
sendingstatemes
counter[myid]==1,Active[myid]==1,Change[myid]==0
tobemessagestate==0
tobesender:=myid,
tobemessagestate:=Sync,
tobemessageposition:=position[myid],
SYNC[myid]:=1
t10!
lifttobus!
countenough
sendingsyncmes
waitforbus1

Fig. 3. Part of the automaton *Station*: Fast loop.