Verified Design of an Automated Parking Garage

Aad Mathijssen and A. Johannes Pretorius

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{a.h.j.mathijssen,a.j.pretorius}@tue.nl

Abstract. Parking garages that stow and retrieve cars automatically are becoming viable solutions for parking shortages. However, these are complex systems and a number of severe incidents involving such garages have been reported. Many of these are related to safety issues in software. We apply verification techniques to develop a software design for an automated parking garage. This design meets a number of safety requirements. We provide a software architecture that allows one to split implementation, safety and algorithmic aspects of the software. Consequently, we give a high-level description of the safety aspects and verify a number of safety requirements on this model. Also, we briefly discuss how this analysis is simplified by using a custom visualization tool.

1 Introduction

Many large cities cope with parking shortages. Traditionally, this has been dealt with by building parking garages below street level or by erecting multi-storey parking arcades. However, large parts of the floor area cannot be used for parking since driving lanes need to be provided. Automated parking garages do not require drivers to park their cars themselves. Instead, cars are placed into parking spaces fully automatically, using a combination of hardware and software. The area needed by such placement mechanisms is usually much less than that needed for driving lanes. This drastically increases parking capacity. Apart from being space efficient, automated parking garages often serve as status symbols for companies or city councils.

Automated parking garages are complex systems. This is reflected by their complex hardware. It is even more evident if one considers some of the incidents involving such systems [1]. These range from users obtaining the wrong car, or no car at all, to cars and equipment being reduced to rubble. The latter is a so-called *safety issue:* the system causes irrecoverable damage to cars or to itself.

In this article we treat safety aspects involved in the software design of a typical automated parking garage. By the time that we were consulted, the hardware design of the investigated system had been finalized. As a result, it is completely fixed and far from optimal regarding safety. This puts an extra burden on the software. Unfortunately, this seems to be a frequent mind-set when designing integrated systems: *"don't worry, we'll make it work with software"*.

The approach we take is to obtain a high-level behavioural description of the system. Safety requirements are verified on each state of this model. By identifying violations of the requirements, we are able to discover shortcomings and improve our specification to ensure safety. Process algebras are well suited for such verified design. We use the new mCRL2 language and toolset [2,3,4] to describe system behaviour and to verify requirements. mCRL2 succeeds and extends μCRL [5,6] with which a number of complex systems have been analysed [7,8,9].

In Sect. 2 we describe the automated parking garage in more detail. Based on this description, we define our goals in Sect. 3. In Sect. 4 we discuss how we conceptually divide the system software into three layers: a hardware abstraction layer, a safety layer and a logical layer. This allows us to concentrate on the safety layer, which we argue is essential for ensuring that the system is safe. In Sect. 5, we describe the design of the safety layer. We follow this with a discussion of implementation issues in Sect. 6. We also describe a simple visualization plug-in for the mCRL2 toolset and the insights we gained by using it. We conclude in Sect. 7.

2 Operational Description

The garage we consider was commissioned by property developers and its hardware designed by a company specialising in automated parking systems. It is due to be installed below street level in the basement of an existing building.

Access to the garage is provided by a vertical lift shaft with a door at street level. A single car can be driven into the lift through this door. When the driver and passengers have exited the car and the lift, the car is automatically lowered to an intermediate level, rotated 180° horizontally, lowered to the basement and stowed using a number of conveyor belts and shuttles. To retrieve a car, the same system of conveyor belts and shuttles is used to bring the car to the lift with which it is raised to street level. Since the car had been rotated before, it now faces the street.

The system provides a number of security and safety checks during check-in and check-out of a car. This includes reading a transponder card and checking a database of registered users before opening the lift door. As the car is driven into the lift, the driver is provided with cues to ensure a correct positioning. There is also a check to ensure that the handbrake is engaged. Before lowering the car to the basement, the lift is scanned to ensure that there are no living beings present.

In the remainder of this article, we consciously abstract from hardware details. We also restrict ourselves to the vertical lift and the basement level parking garage. We do not consider the mechanisms put in place for regulating traffic outside the lift, correctly positioning the car in the lift, or cues to enter and leave the lift. We do this to tightly draw the bounds of our scope. It also allows us to focus on the most important safety aspects of the system.

We assume the operation of the system is initiated every time a car is positioned appropriately in the lift at street level or when a request for retrieving a car is received. The lift can be in one of three vertical positions: street level, rotation level or basement level. At rotation level, the lift is able to rotate 180° horizontally, provided that there are no cars positioned immediately adjacent to the lift shaft (on either side) at basement level. The floor of the lift consists of a conveyor belt. When the lift is at the basement level this conveyor belt is able to move sideways (see the description below). The most complex and interesting part of the system is the basement level. Here the movement of cars is facilitated by a number of conveyor belts and shuttles. This is illustrated in the floor plan of the basement in Fig. 1.

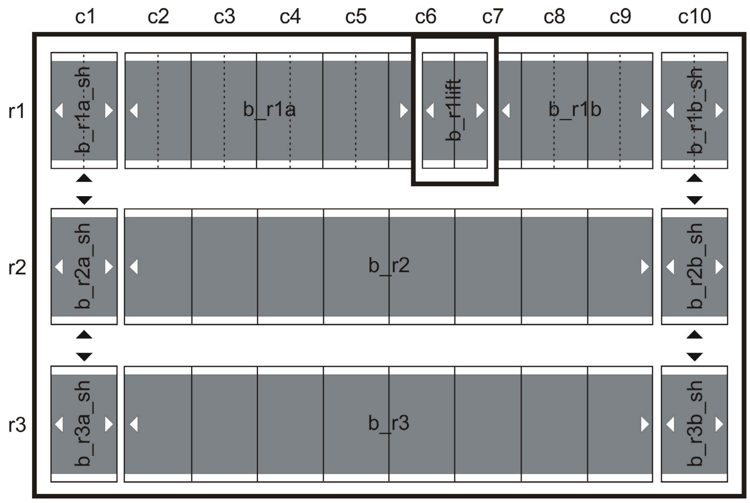


Fig. 1. Floor plan of the parking garage, basement level

As shown in Fig.1, the garage is divided into three rows (r1, r2 and r3) and ten columns (c1–c10). Conveyor belts are represented by grey rectangles with arrows on their ends and are identified by labels such as *b_r1a_sh*. The arrows indicate their direction of movement. Columns c1 and c10 contain three shuttles each. In each of these columns one shuttle may be tilted upwards on its long end facing the wall. This results in an open position to which adjacent lowered shuttles may be moved. A tilted shuttle may also move to a new row position behind lowered shuttles (this implies that it is possible for two shuttles to be in the same row and column position, provided that one is tilted and the other lowered). Black arrows indicate the directions in which shuttles can move. Similar to the lift, every shuttle contains a conveyor belt that can move sideways.

The lift shaft is in row r1. Notice that it is not placed over a full position, but intersects two columns (c6 and c7). This is due to the construction of the building in which the garage is to be installed (and beyond the control of the engineers who designed the parking installation). More importantly, this implies that it must be possible to move cars half-column distances in the first row. For this reason, every column in row r1 is also divided into an *a* (left) and *b* (right) part as indicated by the dashed lines in Fig. 1. We use this same convention in naming the conveyor belts. Hence, *b_r1a_sh* refers to the conveyor belt of the shuttle on the left-hand side of row r1, and so forth.

It is possible for adjacent conveyor belts to move simultaneously to function as a single larger conveyor belt. For instance, *b_r3a_sh* and *b_r3* can be moved in unison. The system hardware can determine whether any (half-)position is free or occupied. For any column in r1, it is possible to determine the status of its *a* (left) and *b* (right) part. It is possible to determine whether there is a lowered, a tilted or no shuttle at all in any row of c1 and c10. Furthermore, the current height of the lift and its status (free or occupied) can be determined.

3 Problem Description

We have mentioned that hardware design is outside the scope of this article. It is also not our goal to develop algorithms. Instead, our aim is to provide specialists in algorithm design with an interface to an abstraction of the underlying hardware that guarantees the safe and correct operation of the system. This provides a clear separation of concerns.

Placement and retrieval algorithms need to ensure that cars are efficiently stowed and retrieved in a fashion that resembles a large sliding puzzle. Even if these algorithms contain errors, the safety interface should not allow the parking garage or the cars in it to get damaged. It needs to specify the necessary checks and restrictions that guarantee the execution of only safe or legal moves. For example, when an algorithm requests that a car be moved to a position that is already occupied by another car, the safety layer should not allow this. The safety interface must also be able to report on the success or failure of issued commands. Properly designed algorithms should be able to respond to such feedback in an appropriate fashion.

4 Conceptual System Design

We now provide a high-level software design for the automated parking garage.

4.1 Architecture

Our aim is to specify a safety interface that sits between placement and retrieval algorithms and the abstract hardware of the automated parking garage. This interface must allow only safe or legal instructions and report on their success or failure. To achieve this, we introduce a three-layered architecture consisting of a logical layer (LL), a safety layer (SL) and a hardware abstraction layer (HAL) (see Fig. 2). With this division into layers, the safety layer ensures the safe operation of the system independently of the particular algorithms that are implemented and without being concerned with hardware implementation issues.

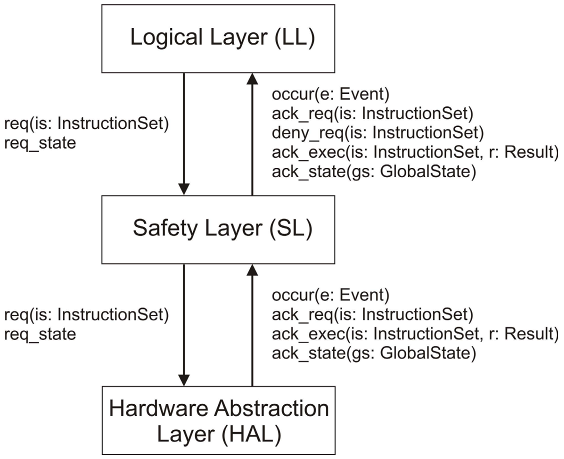

Logical Layer (LL)
req(is: InstructionSet)
req_state
occur(e: Event)
ack_req(is: InstructionSet)
deny_req(is: InstructionSet)
ack_exec(is: InstructionSet, r: Result)
ack_state(gs: GlobalState)
Safety Layer (SL)
req(is: InstructionSet)
req_state
occur(e: Event)
ack_req(is: InstructionSet)
ack_exec(is: InstructionSet, r: Result)
ack_state(gs: GlobalState)
Hardware Abstraction Layer (HAL)

Fig. 2. Three-layered architecture

Data. The following data are communicated between the layers:

– *Event*: an event outside the scope of our design that has an impact on the system. We identify the following events:
  - *add_car*: a new car enters the lift.
  - *remove_car*: a car is removed from the lift.
– *InstructionSet*: a set of instructions to be executed concurrently by the HAL. It consists of a number of elements of type *Instruction*. The notion of a set of instructions allows for the execution of multiple instructions that apply to non-overlapping areas of the basement. For instance, with a single request it is possible to issue different instructions for moving conveyor belts as long as the belts in question do not overlap (see next section).
– *Instruction*: a single instruction that the hardware should execute. Instructions should be implemented by the HAL (see Sect. 4.2). There are 5 different instructions:
  - *move_belts(bs: BeltSet, d: Direction, ms: MoveSize)*: move the set of belts *bs* in direction *d* by a distance of size *ms* (half or full position).
  - *move_shuttles(shs: ShuttleSet, o: ShuttleOrientation, d: Direction)*: move the set of shuttles *shs* in orientation *o* (lowered or tilted upwards) in direction *d* by a distance of one row interval.
  - *tilt_shuttle(p: ShuttlePosition, o: ShuttleOrientation)*: lower or tilt the shuttle in positions *p* and orientation *o* to the orientation that is the opposite of *o*.
  - *move_lift(h: Height)*: move the lift to vertical position *h*.
  - *rotate_lift*: rotate the lift 180° horizontally.
– *Result*: indicates whether a set of instructions has been executed successfully (*ok*) or whether it has failed (*fail*).
– *GlobalState*: the current system state. That is, for every position whether it is free or occupied (*FloorState*), for every shuttle whether it is lowered or tilted upwards (*ShuttleState*), and for the lift its current vertical position and also whether it is free or occupied (*LiftState*).

Interactions. To facilitate communication between the different layers, we introduce the following interactions:

– *occur(e: Event)*: the occurrence of an event *e*. When the HAL detects *e*, the SL is informed by the action *occur(e)*. The SL informs the LL by issuing a similar *occur(e)* action. For the sake of design modularity, the LL is not directly informed by the HAL.
– *req(is: InstructionSet)*: request the execution of instructions *is*. A request from the LL to execute instructions is propagated to the HAL via the SL.
– *ack_req(is: InstructionSet)*: a signal from the SL that the instructions in *is* are safe.
– *deny_req(is: InstructionSet)*: a signal from the SL that the instructions in *is* are unsafe. The SL makes no subsequent requests to the HAL.
– *ack_exec(is: InstructionSet, r: Result)*: the instructions in *is* have been completed with result *r*. The HAL issues an *ack_exec(is, r)* action to the SL. Consequently, the SL issues this action to the LL.
– *req_state*: request the current global state of the system.
– *ack_state(gs: GlobalState)*: communicate the current global state from the HAL to higher layers. This is done in response to a *req_state* action.

4.2 Hardware Abstraction Layer (HAL)

We assume that the HAL serves as a coherent interface to all individual hardware components in the garage. The HAL receives requests for sets of instructions from the SL (action *req*). For each set of instructions the HAL attempts to execute the individual instructions and reports back on the result of the attempt (action *ack_exec*). For the specification of the SL to yield the intended results, instructions must be implemented correctly by the HAL. We assume that this is the case. Apart from executing instructions, the HAL also provides the SL with the current system state before and after issuing any instructions. Such a system state is constructed by the HAL using sensors that monitor the status of the conveyor belts, the shuttles and the lift.

4.3 Logical Layer (LL)

The LL allows for the development of algorithms by experts who can plug their specifications into the SL. These algorithms may contain errors that trigger the request of unsafe instructions. Due to the safety layer, such requests are harmless and will be blocked. More concretely, the LL utilises the SL by requesting that sets of instructions be executed. The SL reports back to the LL in the form of *ack_req*, *deny_req* and *ack_exec* actions. It also informs the LL of events that have occurred (a new car to stow or the removal of an existing car). The logical layer should respond to such feedback appropriately.

4.4 Safety Layer (SL)

The SL sits between the LL and the HAL. It receives requests (*req*) from the LL, which it acknowledges (*ack_req*) or denies (*deny_req*), based on whether the instruction is safe for the current state. After an acknowledgement is sent, the request is passed on to the HAL. In other cases the SL passes information between the HAL and the LL. We treat the SL in more detail in the next section.

5 Verified Design of the Safety Layer

In this section, we identify the safety requirements for the SL and we develop a specification in mCRL2, for which these should be satisfied. Consequently, we verify that this is indeed the case. A more detailed account of the analysis can be found in the corresponding technical report [10]. The full mCRL2 specification and verification code is also included in the report.

5.1 Informal Requirements

The SL should meet the following safety requirements:

1. Conveyor belts:
   (a) When a car is moved from one belt to another, both belts should move in the same direction.
   (b) Cars should never be moved into walls.
   (c) Cars should never be moved to a belt that is not available (when a shuttle is tilted, or when the lift belt is not at the basement level).
2. Shuttles:
   (a) Shuttles should never be moved into a wall.
   (b) When moving shuttles, no shuttles should be damaged.
   (c) When moving shuttles, cars should not be damaged.
   (d) When tilting a shuttle, no shuttles should be damaged.
   (e) When tilting a shuttle, cars should not be damaged.
3. Lift:
   (a) When moving the lift, cars should not be damaged.
   (b) When rotating the lift, cars should not be damaged.

5.2 Specification of Data Types

The data types described in Sect. 4.1 can be translated into mCRL2 in a relatively straightforward way:

```
sort
  Instruction    = struct move_belts(R1BeltSet, DirCol, MoveSize)
                        | move_belts(R2BeltSet, DirCol, MoveSize)
                        | move_belts(R3BeltSet, DirCol, MoveSize)
                        | move_shuttles(ShuttlePosSet, ShuttleOrientation, DirRow)
                        | tilt_shuttle(ShuttlePos, ShuttleOrientation)
                        | move_lift(LiftHeight)
                        | rotate_lift;
  InstructionSet = List(Instruction);% representing a set of instructions
  Event          = struct add_car | remove_car;
  Result         = struct ok | fail;
  GlobalState    = struct glob_state(fs : FloorState, shs : ShuttleState, ls : LiftState);
```

The ambiguity of the definition of *Instruction* is resolved by the fact that *R1BeltSet, R2BeltSet* and *R3BeltSet* are different types. These types are defined in fashion similar to the above definition. Note that the data language of mCRL2 is fully higher-order and we use this to implement some data types as functions. For instance, the datatype *FloorState* is defined as *FloorPos* → *OccState*. That is, a function from floor positions to occurrence states indicating whether a position is free or occupied.

Allowed Instruction Sets. The informal requirements introduced in Sect. 5.1 apply to the *InstructionSet* data type. In our specification of the SL we use an *allowed* function on sets of instructions to determine whether they are safe. A set of instructions *is* is allowed if:

1. *is* specifies at least one instruction.
2. The instructions in *is* do not *overlap*. That is, the positions on which the instructions operate are pairwise disjoint. Consequently, it is safe to execute the instructions in *is* simultaneously.
3. Each individual instruction in *is* is allowed.

Formulated in mCRL2:

```
map
   allowed : InstructionSet × GlobalState → Bool;
   % allowed(is, gs) indicates if instruction set is is allowed given global state gs
   allowed : Instruction × GlobalState → Bool;
   % allowed(i, gs) indicates if instruction i is allowed given global state gs
   overlap : Instruction × InstructionSet → Bool;
   % overlap(i, is) indicates if instruction i overlaps with any of the instructions in is
var
   gs : GlobalState;
   i, j : Instruction;
   is : InstructionSet;
eqn
   allowed([], gs)           = false;
   allowed(i ▷ [], gs)       = allowed(i, gs);
   allowed(i ▷ j ▷ is, gs) = allowed(i, gs) ∧ ¬overlap(i, j ▷ is) ∧ allowed(j ▷ is, gs);
```

Here × represents the Cartesian product and ▷ represents the list cons operation. We are now left with the task of defining the functions *overlap* and *allowed*. We elaborate on the latter, which is the least trivial.

Allowed Instructions. The core of the *allowed* function on instruction sets is an *allowed* function on individual instructions. We describe this function for every instruction.

move_belts(bs: BeltSet, d: Direction, ms: MoveSize) is allowed if:

1. *bs* contains at least one conveyor belt.
2. All conveyor belts in *bs* directly border each other (this also implies that they must be in the same row).
3. All conveyor belts in *bs* are available (this applies to belts on the lift and on shuttles).
4. At least one position of size *ms* (full or half) is free at the end of the set of belts specified. This free position is on the side indicated by *d*.
5. If the specified belts are in row r1, there is no car with one half on a belt in *bs* and the other half on a neighbouring belt not in *bs*.

move_shuttles(shs: ShuttleSet, o: ShuttleOrientation, d: Direction) is allowed if:

1. *shs* contains at least one shuttle.
2. All shuttles in *shs* border each other (this implies that they are all in column c1 or c10).
3. All specified shuttles are available in the orientation specified by *o* (lowered or tilted).
4. There is an open position at the end of *shs* in orientation *o* and direction *d*. This ensures that there is an open position for the shuttles to move to.
5. For every lowered r1 shuttle *s* in *shs*, there is no car with one half on *s* and the other half on a neighbouring belt.

tilt_shuttle(p: ShuttlePosition, o: ShuttleOrientation) is allowed if:

1. It is not the case that there is both a lowered and a tilted shuttle at the position specified by *p*.
2. If *o* is lowered, there is no car on the shuttle (fully or partially).

move_lift(h: Height) is allowed if:

1. *h* is not the current height.
2. If the current height is basement level, there are no cars with one half on the lift and the other half on a neighbouring belt.

rotate_lift is allowed if:

1. The lift is at rotation level.
2. Three half-positions on both sides of the lift are free. This prevents cars in these positions from being damaged by the rotation mechanism.

As an illustrative example of how the *allowed* function has been formally defined, we provide its definition for the *rotate_lift* instruction below. The complete formal specification of the *allowed* function can be found in [10].

```
var
  fs : FloorState;
  shs : ShuttleState;
  ls : ListState;
eqn
  allowed(rotate$_l$ift, glob$_s$tate(fs, shs, ls)) =
    height(ls) $\approx$ rotate $\wedge$
    free([pos_r1(c5, pa), pos_r1(c5, pb), pos_r1(c6, pa),
         pos_r1(c7, pb), pos_r1(c8, pa), pos_r1(c8, pb)], fs);
```

Here ≈ denotes the equality function on data types.

5.3 Specification of Behaviour

For the specification of the behaviour of the SL we note the following:

– At any point in time, the SL processes a single set of instructions. Multiple sets of instructions would complicate the system without having performance benefits (instead of buffering instruction sets in the LL, they would also have to be buffered in the SL).

– We do not take the message passing of the system state or external events into account, as this does not impact safety.

The interactions of Sect. 4.1 are specified by splitting them into *actions* representing the send and receive parts, as illustrated below. The *Layer* parameters indicate the sending and receiving layer of the action, respectively.

```
sort
  Layer = struct logical | safety | hardware;

act
  snd_req, rcv_req : Layer × Layer × InstructionSet;
  snd_ack_req, rcv_ack_req : Layer × Layer × InstructionSet;
  snd_deny_req, rcv_deny_req : Layer × Layer × InstructionSet;
  snd_ack_exec, rcv_ack_exec : Layer × Layer × InstructionSet × Result;
  snd_state, rcv_state : Layer × Layer × GlobalState;
  snd_event, rcv_event : Layer × Layer × Event;
```

Finally, using the actions defined above, the behaviour of the SL is specified as follows:

```
sort
  ProcState = struct ps_idle | ps_ack_deny | ps_req | ps_exec | ps_ack_exec;

proc
  SL(ps : ProcState, gs_sl : GlobalState, is : InstructionSet, r : Result) =
    (ps ≈ ps_idle) →
      (Σ_{isa:InstructionSet} valid(isa) → rcv_req(logical, safety, isa) ·
        SL(ps_ack_deny, gs_sl, isa, r)
      )
  +
    (ps ≈ ps_ack_deny) →
      (allowed(is, gs_sl) →
        (snd_ack_req(safety, logical, is) · SL(ps_req, gs_sl, is, r)) ◇
        (snd_deny_req(safety, logical, is) · SL(ps_idle, gs_sl, is, r))
      )
  +
    (ps ≈ ps_req) →
      snd_req(safety, hardware, is) · SL(ps_exec, gs_sl, is, r)
  +
    (ps ≈ ps_exec) →
      (Σ_{ra:Result} rcv_ack_exec(safety, hardware, is, ra) ·
        SL(ps_ack_exec, nextstate(is, ra, gs_sl), is, ra)
      )
  +
    (ps ≈ ps_ack_exec) →
      snd_ack_exec(safety, logical, is, r) · SL(ps_idle, gs_sl, is, r)
  +
    (ps ≈ ps_idle) →
      (Σ_{e:Event} possible(e, gs_sl) → rcv_event(hardware, safety, e) ·
        SL(ps, nextstate(e, gs_sl), is, r)
      )
  ;

init
  SL(ps_idle, init_gs, [ ], ok);
```

This specifies a process SL with 4 parameters, representing the current state of the process (*ps*) and the garage (*gs_sl*), and the instruction set (*is*) and execution result (*r*) that are to be processed. The behaviour is a collection of alternatives formed from condition-action-result sequences. A summation over a data type indicates a choice over all elements of that data type. Finally, we use three additional functions: *valid* ensures that the lists we use to model sets do not contain duplicates, *possible* indicates whether it is possible for an event to occur, and *nextstate* returns the new state of the system. We do not provide specifications of these functions here.

5.4 Reductions

Due to the enormous number of possible instruction sets that can be requested and executed, it is impossible to perform simulation, let alone verification, on the behavioural specification. This section describes a number of reductions we apply to enable verification. The corresponding specifications can be found in [10].

Reduction 1. Abstract from sets of instructions by focusing on single instructions only.

On the one hand this abstraction is dangerous, because sets of instructions are an essential part of the system. On the other hand, the core safety issue lies in the *allowed* function applied to single instructions. Furthermore, the number of possible system configurations remains the same, since the result of executing a set of instructions concurrently is the same as executing them sequentially. This implies that in the corresponding state space the number of states remains fixed, but the number of transitions is reduced substantially.

Although the former reduction makes it possible to perform simulation, it is not very effective. The aim is to focus on logical mistakes and not hardware failures. For this reason, we also abstract from non-essential messages.

Reduction 2. Abstract from requests and acknowledgements. It is assumed that instructions are executed successfully by the HAL.

The state space corresponding to the specification after applying the above reductions is still prohibitively large. It consists of a calculated total of 6.4×10^{11} (640 billion) states, and a multiple of this in transitions. Hence, we apply one last abstraction.

Reduction 3. The number of positions of the belts is reduced to the minimum that retains the behavioural characteristics of the original configuration. This entails the following. The positions on the conveyor belts *b_r2* and *b_r3* are reduced to two positions each (see Fig. 3). Also, belts *b_r1a* and *b_r1b* are reduced to $1\frac{1}{2}$ full positions (or 3 half positions) each.

The resulting state space has 3.3×10^6 (3.3 million) states and 9.8×10^7 (98 million) transitions, which existing verification tools can manage. Although strictly speaking we are not concerned with proving deadlock-free behaviour, we note that this state space contains no deadlock.

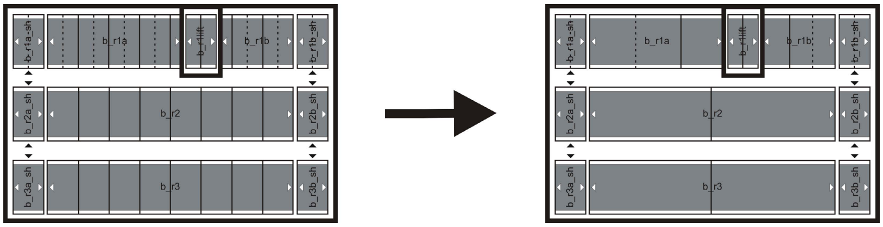


Fig. 3. Reduction of the floor plan

5.5 Formal Requirements and Verification

We verify our safety requirements by extending the specification with *error* actions that are only executed when a requirement is violated. Hence, the requirements are fulfilled when the state space does not contain any error actions. To do this, we translate the high-level requirements from Sect. 5.1 to a lower level of detail. For example, requirement 3(a) is translated to the following.

For all heights h the instruction *move_lift(h)* should not be allowed if both:

1. The lift is at the basement level.
2. The lift contains a car placed halfway on the lift.

This is translated to mCRL2 as the condition-action-result sequence shown below, and is appended to the original specification.

$$\begin{array}{l}
\sum_{\mathrm{i:Instruction,lh:LiftHeight}}(\mathrm{i} \approx \mathrm{move\_lift(lh)} \wedge \mathrm{valid(i)} \wedge \mathrm{allowed(i, gs\_sl)} \wedge \\
\quad \mathrm{ls(gs\_sl)} \approx \mathrm{ls\_basement} \wedge \\
\quad \neg\mathrm{free(positions(b\_r1lift), fs(gs\_sl))} \wedge \\
\quad \neg(\mathrm{even\_occ(positions\_b\_r1([b\_r1a\_sh, b\_r1a]), fs(gs\_sl))} \wedge \\
\qquad \mathrm{even\_occ(positions\_b\_r1([b\_r1b, b\_r1b\_sh]), fs(gs\_sl)))} \\
\quad) \rightarrow \mathrm{error(req3a, 1)} \cdot \delta
\end{array}$$

Extra care is taken to specify the enabling conditions of the error actions: the use of elements of the definition of the *allowed* function are avoided as much as possible, since mistakes in the original specification could carry over to the verification. When we extend the original specification in this fashion, it does not contain any errors. This means that all the requirements are fulfilled.

We conclude this section with some figures. The complete specification contains 991 lines of mCRL2 code, whereas the verification code contains 217 lines of mCRL2, amounting to a total of 1208 lines. Verification took 35 hours and 16 minutes on a single PC (3 GHz CPU, 4 GB RAM), and 5 hours and 38 minutes on a cluster of 34 CPUs (3 GHz CPU, 2 GB RAM). The specification and analysis of the safety layer required approximately 480 man hours to complete.

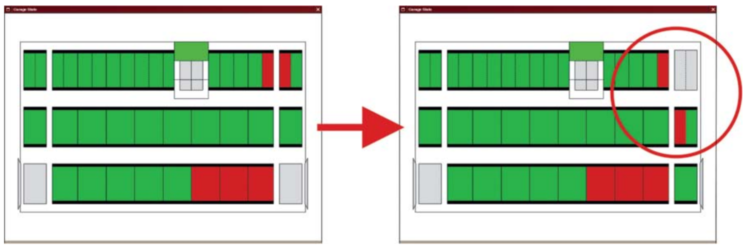
6 Discussion

Before we draw conclusions, we elaborate on two issues. We discuss how visualization helped us during the analysis. Also, we mention some issues regarding the implementation of software based on our specification.

6.1 Visualization

During specification we often resorted to simulating the behaviour of the system using the mCRL2 toolset. The simulator tool allows us to quickly and incrementally check whether our specification results in the behaviour we had anticipated. This is opposed to generating and examining an entire state space which is quite a time consuming undertaking. However, we soon realised that interpreting the text-based output of the simulator is arduous, not entirely intuitive, and prone to human error.

To address these problems and inspired by other visualization initiatives for systems analysis [11,12], we implemented a very simple visualization tool as a plug-in to the simulator. This tool receives the current system state from the simulator and maps it onto a simple 2D floor plan of the parking garage (see Fig. 4). The visualization uses visual cues to indicate the vertical lift position and whether a specific position is *occupied* (red or dark grey), *free* (green or medium gray) or *unavailable* (light gray). Tilted shuttles are also shown. After selecting a new transition, the visualization is updated and the user is able to analyse the system using this representation. The plug-in is distributed with the mCRL2 toolset [4].


(a) Shuttle-induced error

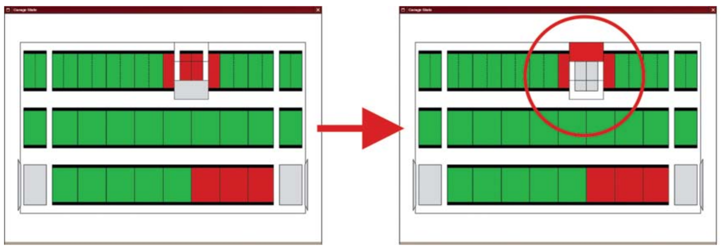

(b) Lift-induced error

Fig. 4. Mistakes identified with the visualization plug-in

Using our visualization tool we discovered a number of problems related to the fact that cars may be moved by half positions in row r1. For instance, we moved a car toward the side of the garage and positioned it with one half on a conveyor belt and one half on a shuttle. To our surprise, it was possible to subsequently move the shuttle, literally tearing the car in half (see Fig. 4(a))! This first bug was relatively easy to fix. A similar problem occurred when two cars were positioned side-by-side on the lift, each with one half on a neighbouring conveyor belt. In this case, despite our best efforts to explicitly check for such a situation, it was possible to move the lift upward, tearing two cars in half (see Fig. 4(b)). This turned out to be a harder problem to solve and involved keeping track of the number of half-positions occupied in row r1 (see Sect. 5.5).

We found the representation of our visualization tool to be intuitively clear and easy to understand. We also believe that this mode of analysis saved a great amount of time. Since we could follow cars as they were transported down the lift and moved to new positions using the conveyor belts and shuttles, we were able to construct potentially dangerous scenarios, such as those discussed above relatively easily. This allowed us to identify and correct a number of problems early on. These included mistakes on our part as well as unknown complexities about the system setup. Although all requirements should also be checked during formal verification, we emphasise that this rests on the assumption that *all relevant questions have been identified and formalised.* By visualizing the current state it is possible to identify issues that may not have been noted otherwise.

6.2 Implementation

As far as software development is concerned, the next step would be to use the specification of data types and behaviour discussed in this article as a starting point for an implementation. Moreover, we believe that reuse of the *allowed* function is of crucial importance. Should software be designed and implemented without considering this, we believe that some of the problems we identified and addressed could easily creep into the implementation despite considerable precaution on the part of the programmers. Unfortunately, we know of cases where such an approach was not taken and where both cars and vital equipment were damaged [1].

Before starting to implement the safety layer, the following aspects require further investigation:

- We have only formally verified the specification for *individual* instructions (see Sect. 5.5). Although we believe that the specification is also correct for sets of instructions, this cannot be guaranteed.
- We do not distinguish between the occurrence of a *recoverable* and an *unrecoverable* hardware failure. All failures are assumed to be recoverable. Furthermore, execution of a set of instructions only gives one result which holds for all instructions. For more detailed error handling, the execution of individual instructions should also return results. That is, after the execution of a set of instructions, some elements may return *ok* while others may return *fail*.
- In practice, the events *add_car* and *remove_car* are not atomic. They need to be split up into a *begin* and *end* part. This results in additional safety requirements. For example, it should be impossible to execute a *move_lift* instruction between the *begin* and *end* part of an event.

Although we have not investigated the *efficiency* of the automated parking garage we foresee a performance challenge in terms of the timely stowing and recovery of cars in practice.

7 Conclusion

We have described the verified design of an automated parking garage. We proposed a system design consisting of three layers: a logical layer, a safety layer and a hardware abstraction layer. We have discussed the verified design of the safety layer. Verification guarantees that in every valid configuration of the system, no damage can occur due to the execution of unsafe instructions. We argue that our work comprises the essence of the system design regarding safety. We recommend that a future implementation should closely follow this design.

With regard to the analysis process, although formal verification is necessary to ensure that requirements are never violated, we want to stress that simulation should not be underestimated. All defects in the specification were found using simulation. Also, simulation allowed us to identify and address interesting behavioural characteristics that would probably not have been included in the requirements otherwise. In particular, we found visually supported simulation to be extremely effective. In this way we were able to gain insight into the system in a way that goes further than simply listing and verifying requirements.

Finally, in this case study software development only started after the hardware design was finished. We argue that both hardware and software experts should be involved in the entire design process to ensure that an optimal solution is found. For this reason, we argue that the current combination of hardware and software is far from optimal.

Acknowledgements

We thank Bas Ploeger and Muck van Weerdenburg for their input and suggestions. Hannes Pretorius is supported by the Netherlands Foundation for Scientific Research (NWO) under grant 612.065.410.

References

1. Verdult, E.: In de prak geparkeerd. De Ingenieur **7** (2005) 32–35
2. Groote, J.F., Mathijssen, A., Van Weerdenburg, M., Usenko, Y.S.: From μCRL to mCRL2: motivation and outline. In: Proc. Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond. BRICS NS-05-3 (2005) 126–131
3. Groote, J.F., Mathijssen, A., Ploeger, B., Reniers, M., Van Weerdenburg, M., Van der Wulp, J.: Process algebra and mCRL2, IPA basic course on formal methods 2006. www.mcrl2.org (2006)
4. mCRL2: mCRL2 homepage (2006) www.mcrl2.org.
5. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In: Algebra of Communicating Processes, Workshops in Computing. (1994) 26–62
6. Groote, J.F., Reniers, M.: Algebraic process verification. In: Handbook of Process Algebra. Elsevier (2001) 1151–1208

7. Fokkink, W., Groote, J.F., Pang, J., Badban, B., Van de Pol, J.: Verifying a sliding window protocol in μCRL. In: Proc. 10th Int'l Conf. Algebraic Methodology and Software Technology. Number 3116 in LNCS, Springer (2004) 148–163
8. Groote, J.F., Pang, J., Wouters, A.G.: Analysis of a distributed system for lifting trucks. J. Logic and Algebraic Programming **55**(1–2) (2003) 21–56
9. Pang, J., Fokkink, W., Hofman, R., Veldema, R.: Model checking a cache coherence protocol for a Java DSM implementation. In: Proc. International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE CS Press (2003)
10. Mathijssen, A., Pretorius, A.J.: Specification, analysis, and verification of an automated parking garage. Technical Report 05-25, Technische Universiteit Eindhoven (2005)
11. Pretorius, A.J., Van Wijk, J.J.: Multidimensional visualization of transition systems. In: Proc. 9th Int'l Conf. Information Visualization (IV05), IEEE CS Press (2005) 323–328
12. Van Ham, F., Van de Wetering, H., Van Wijk, J.J.: Interactive visualization of state transition systems. IEEE Transactions on Visualization and Computer Graphics **8**(4) (2002) 319–329