Specification, Analysis and Verification of an Automated Parking Garage

Aad Mathijssen, 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

In this report we discuss the specification, analysis and verification of an automated parking garage in mCRL2, a process algebra with data. We view the parking garage as a system that we conceptually divide into three layers: a logical layer, a safety layer and a hardware abstraction layer. This allows us to abstract from implementation details (hardware abstraction layer) and algorithm design (logical layer). Instead, we are able to focus on the specification of a communication interface between these two layers that only allows safe system behaviour. This interface constitutes the safety layer. For the safety layer, we identify and formulate a number of requirements. These are verified and reported on. We also discuss the analysis of the specification of the safety layer with a simple custom visualization tool. This tool was implemented as a plug-in to the mCRL2 toolset and helped us gain important insights during the specification and analysis of the safety layer.

1 Introduction

During the past two decades process algebras have become increasingly popular tools for describing complex systems that interact with their environment [5]. Such systems differ from classic batch processes that receive input, process the input, produce output and then terminate. Instead, their behaviour is continuously influenced by information that they receive from their environment. An automated parking garage is a prime example of such a system. It is in operation 24 hours a day, 7 days a week: it is continuously on standby, ready to stow away and retrieve cars for users. Furthermore, the state of the garage is constantly changing as cars are added and removed. At any point in time, this state influences which operations are possible and how they are executed.

Process algebra allows for high-level descriptions of systems that interact with their environment. A system is regarded as a number of interacting processes that together describe its behaviour. In this way a mathematical model is acquired and this is used to prove various properties of the system using mathematical proof techniques. The process algebra used in this report is mCRL2 [3]. It succeeds and extends μCRL [4, 5].

With μCRL a large number of real-world systems have been analysed and verified. Some of the successes that μCRL has booked include the detection of an unknown deadlock in the most complex variant of the sliding window protocol [1, 11], the identification of a number of errors in a distributed lifting system for trucks [2] and the discovery of two errors in a Java distributed memory implementation [9]. From all these experiments we unfortunately have

had to formulate the *100% rule*: in 100% of the cases considered, systems turn out to contain more or less serious errors, which are not very hard to detect.

An important characteristic of mCRL2 and its predecessor μCRL is the inclusion of data. Experience has shown that in real-world systems data is of paramount importance [6]. In addition to the execution of actions, it is often the case that data is stored and communicated, thus having a significant influence on system behaviour.

In the remainder of this report we discuss the specification of an automated parking garage in mCRL2. We also discuss the analysis and verification of the system and the techniques used. In section 2 we provide more details on the challenges presented by this system. In section 3 we outline the approach we have taken to address these issues and we explain how we conceptually divide the system into three layers. This allows us to concentrate on one layer, the safety layer, which we believe is essential in verifying that the system is safe. We follow this by a discussion of mCRL2 and the mCRL2 toolset in section 4. In sections 5, 6 and 7 we specify the hardware abstraction layer, the safety layer and the logical layer respectively. We treat the hardware abstraction layer and the logical layer tersely but reveal more details of the safety layer: we provide a specification, formulate a number of safety requirements and verify these. In section 8 we describe a simple visualization plug-in for the mCRL2 toolset as well as the insights we gained from using it. Finally, we draw conclusions in section 9.

2 Problem description

The parking garage that we are concerned with was commissioned by developers in Bremen, Germany and was designed by the Dutch company *CVSS Automated Parking Systems*. It will be realised below street level in the basement of an existing building. Access to the garage will be provided with a vertical lift shaft that has a doorway at street level. To use the facility, users will drive their car through this doorway into the lift. After they have exited from their car and the lift, their car will automatically be lowered to an intermediate level, rotated 180° horizontally, lowered to the basement and stowed away using a number of conveyor belts and shuttles. When a user wishes to retrieve a car, this same system of conveyor belts and shuttles will be used to bring the car to the lift from which it will be brought to street level. Since the car had been rotated before, it will now face the direction of the street. The user steps into the car and drives away through the doorway.

The system will provide a number of security and safety checks during check-in and check-out of a car. This includes reading a transponder card on the car and checking a database of registered users before opening the doorway to the lift. As the car is driven into the lift, the user will be provided with a number of cues to ensure that the car is positioned appropriately. There will also be a check to ensure that the handbrake is engaged. Before lowering the car to the basement, the lift will be scanned to ensure that there are no living beings present. When a user wishes to retrieve a car there will be the necessary security checks to prevent theft. Furthermore, there will be traffic lights outside the doorway to prevent traffic jams and the installation will be provided with driver motors, position sensors, closed circuit television cameras, smoke alarms and sprinklers.

In subsequent descriptions we consciously abstract from details such as driver motors, position sensors and so forth. We also restrict ourselves to only the vertical lift and the basement level parking garage. We do not take into account 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. This is done in order to tightly draw the bounds of our scope and to focus on what we believe are the essential elements that facilitate the safe behaviour of the system. In doing so, we avoid getting bogged down by implementation issues, hardware or logistics.

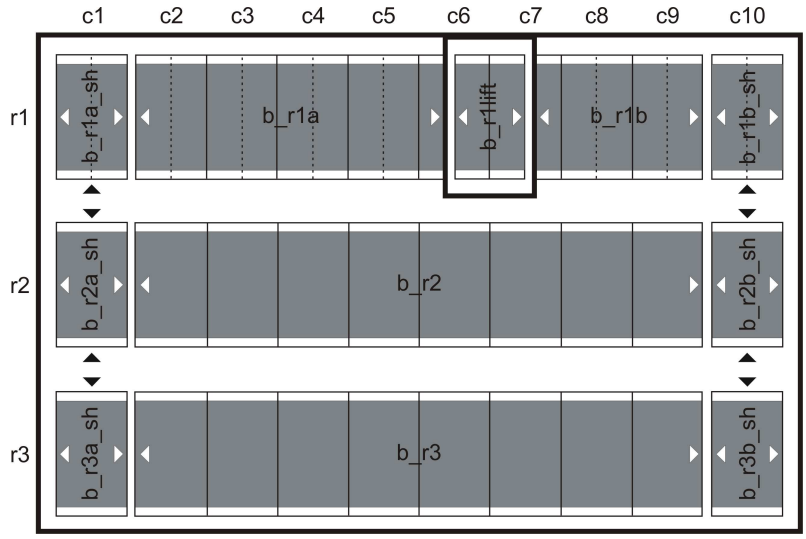

Figure 1: Floorplan of the parking garage, basement level

From our perspective 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 a car is received. Physically we view the system as consisting of a single lift and a parking garage. The lift can be in one of three vertical positions: street level, rotation level or basement level. At the rotation level, the lift is able to rotate 180° horizontally. This is 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 most interesting part of the system is the parking garage at basement level. The movement of cars at this level is facilitated by a number of conveyor belts and shuttles. This is illustrated in the floorplan of the basement in figure 1.

As shown in the figure, the garage is divided into three rows (r1, r2 and r3) and ten columns (c1-c10). Conveyor belts are portrayed by grey rectangles with arrows on their ends and are identified by labels such as *b_r1a_sh* (below we elaborate further on the naming convention used). 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 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). In Figure 1, black arrows indicate the directions in which shuttles can move. Similar to the lift, every shuttle also 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 rather intersects two columns (c6 and c7). This artefact is due to the construction of the building in which the garage is to be installed (and hence beyond the control of the engineers who designed the garage). 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 figure 1. We use this same

convention in the naming of the belts in the figure. 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 to move any subset of conveyor belts and any subset of lowered shuttles. It is also possible to connect neighbouring conveyor belts to function as a single larger conveyor belt. For instance, conveyor belt *b_r3a_sh* can be connected to *b_r3*. Furthermore, one shuttle in c1 and one shuttle in c10 can be tilted. A tilted shuttle can be moved independently from the lowered shuttles in that column.

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* and *b* part. Note that this implies that individual cars on r1 cannot be identified by hardware sensors. Since we only want to guarantee safe behaviour, this does not impact our specification. It is also 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 can be determined and also whether it is free or occupied. This makes it possible to get a snapshot of the system at any point in time during operation.

Apart from not being involved in hardware design, it is also not our goal to be concerned with algorithm design. Instead, our goal is to provide specialists in algorithm design with an interface to an abstraction of the underlying hardware that guarantees the safe and correct operation thereof. This provides a clear separation of concerns. The algorithms need to ensure that cars are efficiently stowed away and retrieved. They must also ensure that the garage can be filled to its maximum capacity of 29 cars (this leaves one position free and allows for the garage to operate in a fashion similar to a large sliding puzzle). Even if there are errors in such algorithms, the interface should not allow the parking garage or the cars in it to get damaged during their execution. It needs to specify the necessary checks and restrictions that guarantee the execution of only safe or legal moves. The safety interface must also be able to report on the success or failure of issued commands. We envision that properly designed algorithms will be able to respond to such feedback in an appropriate fashion.

3 Conceptual system design

3.1 Architecture

As already alluded to above, our aim is to specify a safety layer that sits between eventual placement and retrieval algorithms and the abstract hardware of the automated parking garage. This layer must allow only safe or legal instructions and report on their success or failure. We therefore introduce a three-layered architecture consisting of a logical layer (LL), a safety layer (SL) and a hardware abstraction layer (HAL) (see figure 2). With this conceptual 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.

3.2 Data

The following data are communicated between the layers:

- *Event*: this data type represents events that are outside the scope of our design, but that do have an impact on the system. We identify the following events:
  - *add_car*: a new car has entered the lift.
  - *remove_car*: a car has been removed from the lift.

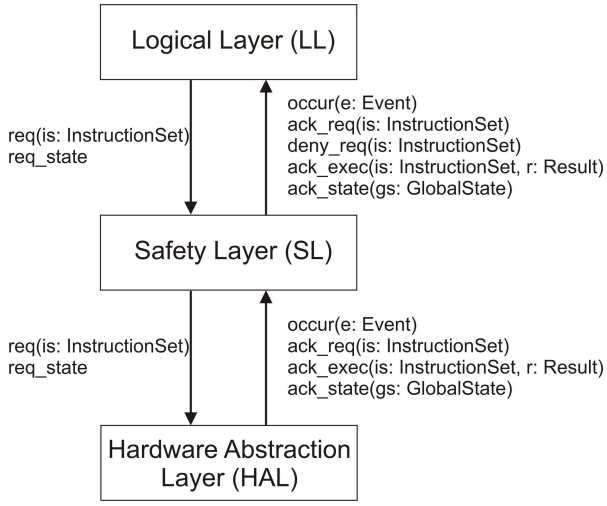

Figure 2: Conceptually the system consists of three layers

- *InstructionSet*: this data type represents sets of instructions that are to be executed concurrently by the HAL. It consists of a number of elements of type *Instruction*.
- *Instruction*: this data type represents the single instructions that the hardware should execute. They should be implemented by the HAL (see section 5). There are 5 different instructions:
  - *move_belts(bs: BeltSet, d: Direction, ms: MoveSize)*: this instruction specifies that the set of belts *bs* should be moved in direction *d* with a distance of size *ms* (half or full).
  - *move_shuttles(shs: ShuttleSet, o: ShuttleOrientation, d: Direction)*: this instruction specifies that the set of shuttles *shs* in orientation *o* (lowered or tilted) should be moved in direction *d* with a distance of one row interval. The indication of orientation *o* is needed for disambiguation, since it is possible for both a lowered and tilted shuttle to be in the same position.
  - *tilt_shuttle(p: ShuttlePosition, o: ShuttleOrientation)*: this instruction specifies that the shuttle in position *p* and orientation *o* should be lowered or tilted to the orientation that is the opposite of *o*.
  - *move_lift(h: Height)*: this instruction specifies a new vertical position *h* to move the lift to.
  - *rotate_lift*: this instruction specifies that the lift should be rotated by 180° horizontally.
- *Result*: this data type indicates whether an instruction has been executed successfully *(ok)* or whether it has failed (*fail*).
- *GlobalState*: this data type reflects 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 (*ShuttleState*), and for the lift its current vertical position and whether it is free or occupied (*LiftState*).

A precise specification of all data types can be found in appendix B.

3.3 Actions

To facilitate communication between the different layers in our conceptual design, we introduce the following actions (also see figure 2):

- *occur(e: Event)*: this action signifies the occurrence of an event *e*. When the HAL detects *e*, the SL is informed by the action *occur(e)*. In turn, the SL informs the LL by issuing an identical *occur(e)* action. Note that for the sake of modularity of the design, the LL is not directly informed by the HAL.
- *req(is: InstructionSet)*: this action allows the LL to request the execution of a set of instructions *is*. This request is propagated to the HAL via the SL. 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, it is possible with a single request to issue different instructions for moving conveyor belts as long as the belts in question do not overlap).
- *ack_req(is: InstructionSet)*: when the LL requests the execution of a set of instructions *is*, and these are deemed safe by the SL, the SL issues an *ack_req(is)* action.
- *deny_req(is: InstructionSet)*: in the case that the SL deems a request for a set of instructions *is* from the LL as unsafe it replies by issuing a *deny_req(is)* action. When this happens, no further requests are made to the HAL by the SL.
- *ack_exec(is: InstructionSet, r: Result)*: upon completion of a set of instructions *is* with result *r*, the HAL issues an *ack_exec(is, r)* action to the SL. In turn, the SL issues this action to the LL.
- *req_state*: this action is used to request the current global state from the HAL via the SL.
- *ack_state(gs: GlobalState)*: this action is used to communicate the current global state from the HAL to higher layers. This is done in response to a *req_state* action.

A precise description of the most relevant actions and their use can be found in appendix C.1.

4 mCRL2

In following sections we provide more details of the actual specification of the system in terms of the layered architecture introduced above. Before we proceed, we introduce the mCRL2 specification language and the accompanying mCRL2 toolset.

The mCRL2 language is a successor to μCRL. The most important improvements on μCRL is that the language is compositional, that is large systems can be specified in terms of smaller components. It also has a more advanced data specification language which is higher-order and provides concrete data types.

Below we only introduce the language features that are used for the specification of the safety layer. The data and process languages are treated separately. A rich text format is used as opposed to plain text in the appendices. To overcome the differences between these formats, a translation table is provided in appendix A. In the plain text format found in the appendices, the %-symbol indicates the beginning of a comment that extends to the end of the line.

4.1 Data language

The mCRL2 data language is in a functional language based on *higher-order abstract data types* [7, 8]. Types, constructors, functions and their definitions can be declared. For instance, the following declares the type A with constructors c, d, functions f, g and the definitions of f, g:

sort A;
cons $c, d : A$;
map $f : A \to A \to A$;
$g : A \to A$;
var $x, y : A$;
eqn $f(c, x) = c$;
$f(d, x) = x$;
$g = f(c)$;

In the equations, *variables* can be used and *pattern matching* on the constructors of the data type is catered. Note that function types are first-class citizens: functions may return functions.

Type references can be declared, for instance in

sort $B = A$;

B is a synonym for A. With this it is possible to define recursive types (see below).

Furthermore, *where clauses* can be used as an abbreviation mechanism, for example:

map $h : A \to A \to A$;
var $x, y : A$;
eqn $h(x, y) = f(z, g(z))$ **whr** $z = g(x, y)$ **end**;

As mentioned above, mCRL2 also has concrete data types. These consist of *standard data types and operations* as well as *type constructors*. For the former, we have the following:

- Booleans ($\mathbb{B}$) with constants *true*, *false* and operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$. For all types the equality operator $==$, inequality $\neq$ and conditional if are provided. So for instance the expression $c == c$ is equal to *true*, $c == d$ to *false* and $if(true, c, d)$ to c. Also, expressions of type $\mathbb{B}$ may be used as conditions in equations, for instance:

  var $x, y : A$;
  eqn $x == y \;\to\; f(x, y) = x$

- Unbounded positive, natural and integer numbers ($\mathbb{P}, \mathbb{N}$ and $\mathbb{Z}$) with relational operators $<, \leq, >, \geq$, unary negation $-$ and binary arithmetic operators $+, -, *, \mathbf{div}, \mathbf{mod}$.

There are two type constructors, of which the first one is a *structured type*. This is a compact way of defining a type together with constructor and projection functions. A sort *MS* of machine states can be declared by:

sort $MS = \mathbf{struct}\ off \mid standby \mid starting \mid running \mid broken$;

The sort of binary trees with numbers as their leaves looks like this:

sort $T = \mathbf{struct}\ \mathit{leaf}(\mathit{value} : \mathbb{N}) \mid \mathit{node}(\mathit{left} : T, \mathit{right} : T)$;

This declares type T with constructors $\mathit{leaf} : \mathbb{N} \to T$ and $\mathit{node} : T \times T \to T$ and projection functions $\mathit{value} : T \to \mathbb{N}$ and $\mathit{left}, \mathit{right} : T \to T$. Functions on T can be defined using pattern matching on the constructors leaf and node.

Finally, there is a *list* type constructor. The following declares a list containing elements of type A;

sort $AL = \mathit{List}(A)$

This list has constructors $[\,] : AL$ and $\triangleright : A \times AL \to AL$. Also operators $\triangleleft$, $+\!\!+$, head, tail, rhead and rtail are provided together with list enumeration $[\_, \ldots, \_]$. The following expressions of type AL are all equivalent: $[c, d, d]$, $c \triangleright [d, d]$, $[c, d] \triangleleft d$ and $[\,] +\!\!+ [c, d] +\!\!+ [d]$.

4.2 Process language

The most basic notion in the mCRL2 process language is an action. Actions represent atomic events. They can be declared in the following way:

act a, b;
$c : \mathbb{B}$;
$d : \mathbb{B} \times \mathbb{P}$;

This declares actions a, b, c and d. Here, a and b are parameterless actions, c is an action with a data parameter of type $\mathbb{B}$, and d has two parameters of type $\mathbb{B}$ and $\mathbb{P}$ respectively. For the above declaration, a, $c(1)$ and $d(6, \mathit{true})$ are valid actions.

Process expressions are compositions of actions using a number of operators:

- Deadlock or inaction δ, which does not display any behaviour.
- Alternative composition, written as $p+q$. This expression non-deterministically *chooses* to execute process expression p or q.
- Sequential composition, written $p \cdot q$. This expression first executes p followed by q.
- Conditional, written as $b \to p, q$, where b is a data expression of type $\mathbb{B}$. The process expression behaves as an if-then-else construct. That is, if b is *true* then p is executed, else q is executed. The else part is optional, that is $b \to p$ is a valid expression that behaves as $b \to p, \delta$.
- Summation over data types. The sum operator $\sum_{d:D} F(d)$, with F a mapping from data type D with constructors $d_0, \ldots, d_n$, $n \geq 0$, to process expressions, behaves as $F(d_0) + \ldots + F(d_n)$.
- Process references, written as $X(d_1, \ldots, d_n)$, with $n \geq 0$. These are references to variables declared by process equations, that are introduced next.

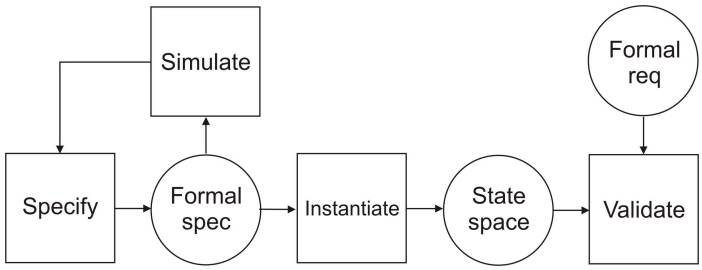

Figure 3: Typical specification, analysis and verification cycle using mCRL2

Using process expressions we can form *process equations*. Take for instance the following declaration, using the action declaration above:

$$\begin{aligned}\textbf{proc}\ \ X(x:\mathbb{B}, y:\mathbb{P}) = &(a+b)\cdot X(x,y)\\ &+ c(x)\cdot X(\neg x, p+1)\\ &+ p>1 \rightarrow b\cdot d(x,p)\cdot X(\mathit{false}, p-1);\end{aligned}$$

This declares process variable X with two data parameters of type $\mathbb{B}$ and $\mathbb{P}$. For data terms $t:\mathbb{B}$ and $u:\mathbb{P}$, $X(t,u)$ behaves as the right-hand side of the equation in which x and y are replaced by t and u. Process variable X is often called a process and parameters t and u are called the state of X.

A process specification needs to be initialised. For example:

$$\textbf{init}\quad X(\mathit{true}, 1);$$

This initialises process X with $(\mathit{true}, 1)$ as its initial state.

4.3 mCRL2 toolset

Specification, analysis and verification of systems with mCRL2 is also supported by a toolset. Some of these tools, and how they fit into the specification, analysis and verification cycle are illustrated in figure 3. Once the specification activity starts, simulator tools allow the user to simulate parts of the resulting state space. Essentially this tool allows the user to follow labelled transitions from one system state to another. At any particular time the simulator provides the user with an overview of the current system state and also provides a listing of all possible transitions from that state (see figure 4). After a transition is selected, the tool updates the system state accordingly, and provides an overview of the new state and all transitions possible from it.

Once users are satisfied with the specification, the instantiator tool generates the corresponding state space. By this stage, requirements should have been formalised and are checked against on the state space to determine whether any states exist that violate these.

5 Hardware abstraction layer (HAL)

We assume that the HAL serves as a coherent interface to all individual hardware components in the garage (driver motors and so forth). 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*).

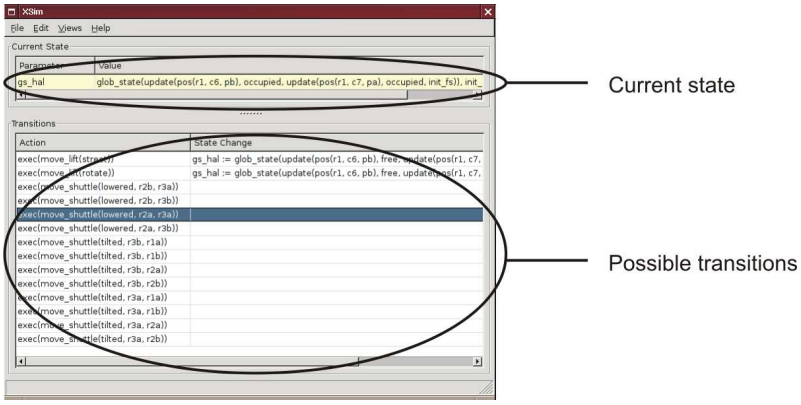

Figure 4: Simulator

In order for the specification of the SL to yield the intended results, instructions must be implemented correctly by the HAL. For instance, the following must occur at hardware level when the instruction *move_belts(bs, d, ms)* is received: connect all the conveyor belts specified in *bs*, engage the appropriate driver motors and move the belts by a distance corresponding to *ms* in the direction indicated by *d*. To ensure that the distance moved corresponds to *ms* the appropriate position sensors need to be utilised.

Apart from executing instructions, the HAL also provides the SL with access to a global snapshot of the current system state before and after issuing any instructions. Such a system state is constructed by the HAL using sensors that monitor the state of every (half-)position of the belts, every shuttle and the lift.

6 Safety layer (SL)

For the safety layer we identify the safety requirements and design a specification for which these should be satisfied. After that we verify that this is indeed the case.

6.1 Requirements

The safety requirements of the safety layer may be summarised as follows:

1. Conveyor belts:
   (a) If a car is moved between belts, both belts should move in the same direction.
   (b) Cars should not be able to move into walls.
   (c) Cars should not be able to move to a belt that is not available (that is a shuttle that is tilted, or the lift belt not being in its lowest position).
2. Shuttles:
   (a) When moving shuttles, a shuttle should not be able to move into the wall.
   (b) When moving shuttles, other shuttles may not be damaged.
   (c) When moving shuttles, cars may not be damaged.

(d) When tilting a shuttle, cars may not be damaged.

(e) When tilting a shuttle, other shuttles may not be damaged.

3. Lift:

   (a) When moving the lift, cars may not be damaged.

   (b) When rotating the lift, cars may not be damaged.

6.2 Specification

In order to verify that the SL allows only safe requests to be propagated to the HAL, we have specified its behaviour in mCRL2. This specification is provided in appendix B and C.1. The following should be noted:

- We did not take the message passing of the system state or the events into account, as this does not impact safety.
- We assume that there is at most one set of instructions in the system. This is not a real restriction, as it seems natural that communication between the layers is much faster than the actual execution of instructions.

The informal requirements introduced above are relevant to the *InstructionSet* data type (see section 5). In our specification of the SL we use an *allowed* function on sets of instructions to determine whether a set of instructions is 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 areas on which the instructions operate are pairwise disjoint such that it is safe to execute the instructions in *is* simultaneously.
3. Each instruction in *is* is allowed.

For the last condition, we also need an *allowed* function on individual instructions, which forms the core of the definition.

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

1. *bs* specifies 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 (in particular, this applies to belts on the lift and on shuttles).
4. At least one position of size *ms* must be free at the end of the set of belts specified, this free position should be on the side indicated by *d*.
5. In the case that the specified belts are in row r1, there must be no car suspended half-way between the two outer belts of *bs* and their neighbours, if any.

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

1. *shs* specifies 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 must be 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 must be no car that is suspended between *s* and its bordering conveyor belt.

The instruction *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 in the position specified by *p*.

2. If *o* is lowered, there is no car on the shuttle (fully or partially).

The instruction *move_lift(h: Height)* is allowed if:

1. The target height specified by *h* is not the current height.

2. If the current height is basement level, there must be no cars suspended half-way between the lift and conveyor belts on either side of the lift.

The instruction *rotate_lift* is allowed if:

1. The current height of the lift is rotation level.

2. Three half-positions on both sides of the lift are free (this to keep cars in these positions from being damaged by the rotation mechanism).

A formal specification of the *allowed* function can be found in appendix B.

6.3 Simulation

Because of the enormous amount of possible instruction sets that can be requested and executed, it is impossible to do any simulation, let alone verification, on the specification in appendix C.1. For this reason we apply the following reduction:

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

The corresponding specification can be found in appendix C.2. 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 on 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 are reduced substantially.

Although the former reduction makes it possible to perform simulation, it is not very effective. For this reason, we abstract from non-essential messages:

Reduction 2 Abtract from requests and acknowledgements. Instead, it is assumed that instructions are executed successfully by the HAL.

The corresponding specification can be found in appendix C.3. When simulating, only allowed instructions can be executed.

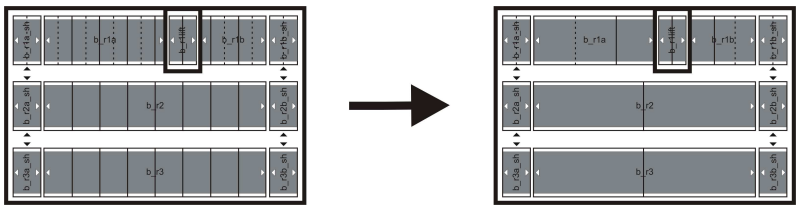

Figure 5: Reduction of the floorplan

6.4 Verification

We want to verify the requirements by exploring the state space. However, the state space corresponding to specification after applying the reduction above consists of $6,4 \times 10^{11}$ (640 billion) states, and a multiple of this in transitions. This is prohibitively large. Hence, we apply one last abstraction:

Reduction 3 The number of positions of the belts are 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 figure 5). Also, belts *b_r1a* and *b_r1b* are reduced to to $1\frac{1}{2}$ full positions (or 3 half positions) each.

The corresponding specification can be found in appendix D.1. Its state space has $3,3 \times 10^6$ ($3,3$ million) states and $9,8 \times 10^7$ (98 million) transitions, which is manageable.

The approach we take in verifying the requirements deviates a little from the typical cycle of figure 3 on page 9. Namely, we *extend* the specification with *error* actions that are only enabled when a requirement is violated. Hence, the requirements are fulfilled when the state space does not contain any error actions.

The specification extended with error actions can be found in appendix D.2. In this appendix a translation from the high-level requirements of section 6.1 to a lower level of detail can also be found. Extra care has been taken to specify the enabling conditions of the error actions. Namely the use of elements of the definition of the *allowed* function needs to be avoided as much as possible, since mistakes in the original specification could carry over to the verification.

Keeping this observation in mind, we have extended the specification and generated the state space. This state space does not contain any error actions, which means that all the requirements are fulfilled.

7 Logical layer (LL)

As mentioned before, the LL utilises the SL by requesting that sets of instructions be executed. In turn, the SL reports back to the LL in the form of *ack_req*, *deny_req* or *ack_exec* actions. It also informs the LL of events that have occurred (a new car to stow away or the removal of an existing car). This 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 instructions are harmless (but should be avoided), since they will be blocked.

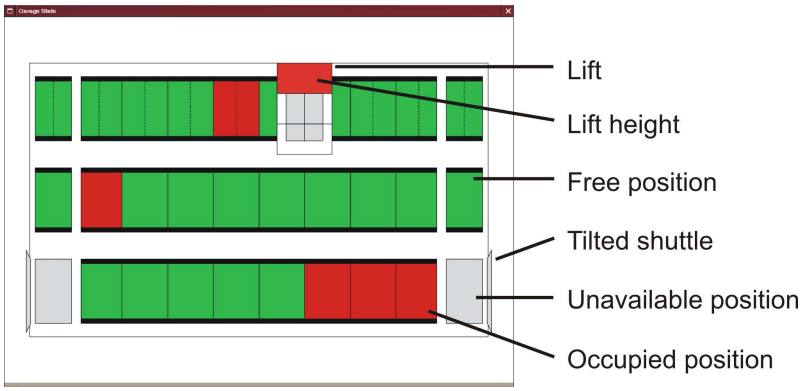

Figure 6: Visualization plug-in

8 Visualization

During the specification we often resorted to simulating the behaviour of the system using the mCRL2 toolset (see section 4.3). The simulator tool is a great aid during specification since it allows us to quickly and incrementally check whether our specification does indeed result in the behaviour that we had anticipated. This is opposed to generating and examining an entire transition system which is quite an expensive undertaking (due to the phenomenon of state space explosion and effort needed for formulating and checking formal requirements). However, during specification we soon realised that interpreting the text-based output of the simulator is time consuming, not entirely intuitive and prone to human error (especially as the amount of information that is stored in states increases).

In order to address the above problem and inspired by other visualization initiatives to aid in system analysis [10, 12], we implemented a very simple visualization tool as a plug-in to the simulator. The visualization tool gets the current system state from the simulator and maps the information onto a simple 2D floorplan of the parking garage (see figure 6). This visualization uses visual cues to indicate the vertical lift position and whether a specific position is *occupied* (red), *free* (green) or *unavailable* (gray). Tilted shuttles are also visualised. Instead of interpreting the text-based representation of the current system state, after selecting a new transition, the visualization is updated and the user is able to analyse the system with this representation.

We believe that this mode of analysis saved a great amount of time. Due to the fact that 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. 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 parking garage setup in general. Although all requirements can be checked during the formal verification stage, this rests on the assumption that all relevant questions have been identified and formalised. By visualizing the current state it is also possible to identify additional issues that may not have been noted. Moreover, had software simply been designed and implemented for the garage without a detailed analysis, we believe that such issues could have easily crept into the implementation of the garage despite considerable precaution on the part of the designers. Unfortunately, we know of a less carefully designed garage where both cars and vital equipment have been damaged.

While using our visualization tool in designing the system we discovered a number of

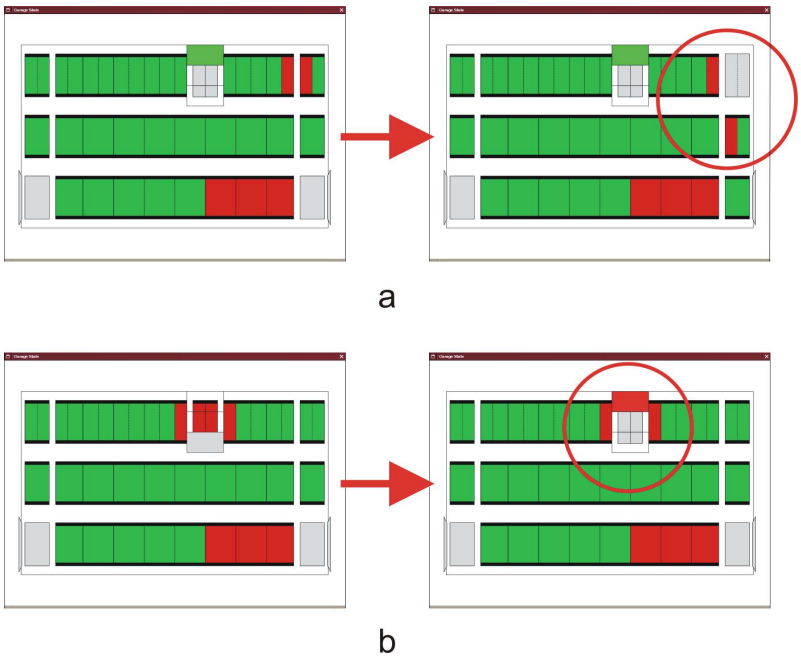

Figure 7: Mistakes identified with the visualization plug-in

problems related to the fact that cars may be moved in half positions in row r1. First, we moved a car toward the sides 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 figure 7a)! This first bug we discovered 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 bordering conveyor belt. In this case, despite our best efforts of explicitly checking for such a situation, it was possible to move the lift up, tearing two cars in half (see figure 7b). This turned out to be a harder problem to solve.

We found the representation of our visualization tool to be intuitively clear and easy to follow. In this case were are fortunate that there exists such an intuitive mapping to a visual representation. This is often not the case. For instance, for communication protocols, there is no "natural" visual representation requiring researchers to come up with more abstract solutions.

9 Conclusions

In this report we have analysed 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 given a formal specification of the safety layer and after applying a number of reductions we were able to verify the safety requirements for this specification. This means that in every valid configuration of the system, no damage can be done by executing unsafe instructions.

We strongly recommend using the specification of data types and processes discussed in this report as a starting point for the implementation of software to operate the parking garage (appendices B and C.1). This holds especially for the conceptual three-layer architecture and

the definitions of the *allowed* function.

We believe that our work comprises the essence of the system design regarding safety. However, the following requires further investigation before starting on the implementation:

- In appendix C.1 a specification of the system is given in terms of *sets* of instructions. Although we believe that the specification is correct for these sets and although we have simulated on some key problem areas regarding sets, we have not been able to verify the safety. This means that extra care should be taken in the implementation of the *allowed* function on sets of instructions in appendix B.
- We do not distinguish between the occurrence of a *recoverable* and an *unrecoverable* hardware failure. All failures are treated as 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 the others may return *fail*.
- In practise, the events *add_car* and *remove_car* are not atomic. They need to be split up in a begin and an end part. In order to guarantee safety, 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 away and recovery of cars in practise. However, we have taken care to ensure that the division into three layers does not degenerate performance. In particular, the notion of the concurrent execution of instructions addresses this issue.

With regard to the general system design, verification and analysis cycle (see figure 3), 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 this way we were able to gain insight into the system in a way that goes further than simply listing and verifying requirements. Still, verification remains necessary.

Finally, we want to point out the danger of using the same data types for both simulation and verification. This is due to the possibility of errors in the data type specification. The approach we took to circumvent this is to use different functions for the verification of the most crucial functions and safety requirements.

Acknowledgements

This work was partially supported by the Dutch Foundation for Scientific Research *(NWO)* under grant 612.065.410. We also wish to thank Muck van Weerdenburg for his valuable input, suggestions and refinement of the mCRL2 toolset.

References

[1] W. Fokkink, J.F. Groote, J. Pang, B. Badban and J.C. van de Pol, "Verifying a sliding window protocol in μCRL", *Proc. 10th Int'l Conf. Algebraic Methodology and Software Technology*, C. Rattray et al., eds., LNCS 3116, Springer Verlag, 2004, pp. 148-163.

[2] J.F. Groote, J. Pang and A.G. Wouters, "Analysis of a distributed system for lifting trucks", *J. Logic and Algebraic Programming*, vol. 55, no. 1-2, 2003, pp. 21-56.

[3] J.F. Groote, A. Mathijssen, M. van Weerdenburg and Y. Usenko, "From μCRL to mCRL2: motivation and outline", *Proc. Workshop Algebraic Process Calculi: The First Twenty Five Years and Beyond*, BRICS NS-05-3, 2005, pp. 126-131.

[4] J.F. Groote and A. Ponse, "The syntax and semantics of μCRL", *Algebra of Communicating Processes, Workshops in Computing*, A. Ponse, et al., eds., 1994, pp. 26-62.

[5] J.F. Groote and M. Reniers, "Algebraic process verification", *Handbook of Process Algebra*, J.A. Bergstra et al., eds., Elsevier Science, 2001, pp. 1151-1208.

[6] J.F. Groote and J.J. van Wamel, "Algebraic Data Types and Induction in μCRL", University of Amsterdam, Amsterdam, Tech. Rep. P9409, 1994.

[7] K. Meinke, "Higher-Order Equational Logic for Specification, Simulation and Testing", *The 1995 Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA '95)*, LNCS 1074, Springer, 1996, pp. 124-143.

[8] B. Möller, A. Tarlecki and M. Wirsing, "Algebraic Specification of Reachable Higher-Order Algebras", *Recent Trends in Data Type Specification*, LNCS 332, Springer, 1988, pp. 154-169.

[9] J. Pang, W. Fokkink, R. Hofman and R. Veldema, "Model Checking a Cache Coherence Protocol for a Java DSM Implementation", *Proc. 2003 International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, IEEE Computer Society Press, 2003.

[10] A.J. Pretorius and J.J. van Wijk, "Multidimensional Visualization of Transition Systems", *Proc. 9th Int'l Conf. Information Visualization (IV05)*, London, IEEE CS Press, 2005, pp. 323-328.

[11] A.S. Tanenbaum, *Computer Networks*, Prentice Hall, 1981.

[12] F. van Ham, H. van de Wetering and J.J. van Wijk, "Interactive visualization of state transition systems", *IEEE Trans. Visualization and Computer Graphics,* vol. 8, no. 3, 2002, pp. 319-329.

Appendix A: Table of mCRL2 symbols

In the specification of appendices below, a plain text format is used as opposed to the rich text format of section 4. To make the translation from rich text symbols to plain text symbols, the following table is provided:

Symbol	*Rich*	`Plain`
Booleans	$\mathbb{B}$	`Bool`
positive numbers	$\mathbb{P}$	`Pos`
natural numbers	$\mathbb{N}$	`Nat`
integers	$\mathbb{Z}$	`Int`
arrow	$\rightarrow$	`->`
inequality	$\neq$	`!=`
logical negation	$\neg$	`!`
conjunction	$\wedge$	`&&`
disjunction	$\vee$	`||`
implication	$\Rightarrow$	`=>`
smaller than or equal	$\leq$	`<=`
greater than or equal	$\geq$	`>=`
list cons	$\triangleright$	`|>`
list snoc	$\triangleleft$	`<|`
list concatenation	$+\!\!+$	`++`
sum	$\sum$	`sum`

Table 1: Translation of mCRL2 symbols

Appendix B: Data type specification

All process specifications use the same data type specification. For this reason, we only list the general part once. This specification is structured as follows. First all data types are declared. After that, functions for each data type are declared and grouped together.

```
sort
  FloorRow = struct r1 | r2 | r3;
  FloorCol = struct c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 | c9 | c10;
  FloorPosPart = struct pa | pb;
  FloorPos = struct pos_r1(FloorCol, FloorPosPart)
                  | pos_r2(FloorCol)
                  | pos_r3(FloorCol);
  FloorPosList = List(FloorPos);
  ShuttlePos = struct r1a | r2a | r3a | r1b | r2b | r3b;
  ShuttlePosSet = List(ShuttlePos); % representing a set of ShuttlePos elements
  ShuttleOrientation = struct lowered | tilted;
  R1Belt   = struct b_r1a_sh | b_r1a | b_r1lift | b_r1b | b_r1b_sh;
  R2Belt   = struct b_r2a_sh | b_r2 | b_r2b_sh;
  R3Belt   = struct b_r3a_sh | b_r3 | b_r3b_sh;
  R1BeltSet = List(R1Belt); % representing a set of R1Belt elements
  R2BeltSet = List(R2Belt); % representing a set of R2Belt elements
  R3BeltSet = List(R3Belt); % representing a set of R3Belt elements
  DirCol = struct col_inc | col_dec;
  DirRow = struct row_inc | row_dec;
  MoveSize = struct full | half;
  LiftHeight = struct street | rotate | basement;
  Area =
    struct area(positions: FloorPosList, tilted_c1: Bool, tilted_c10: Bool);
  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;
  ExecResult = struct ok | fail;
  OccState = struct free | occupied;
  AvailState = struct avail | n_avail;
  FloorState = FloorPos -> OccState;
  ShuttleState = ShuttlePos # ShuttleOrientation -> AvailState;
  LiftState = struct lsf_street | lso_street
                   | lsf_rotate | lso_rotate
                   | ls_basement;
  GlobalState =
    struct glob_state(fs: FloorState, shs: ShuttleState, ls: LiftState);

% FloorRow operations
map
  index: FloorRow -> Pos;
  % index(fra) is the index of fra
  lt: FloorRow # FloorRow -> Bool;
  % lt(fra,frb) indicates if fra is less than frb
var
  fra, frb: FloorRow;
eqn
  index(r1) = 1;
  index(r2) = 2;
  index(r3) = 3;
  lt(fra, frb) = index(fra) < index(frb);

% FloorCol operations
map
  index: FloorCol -> Pos;
  % index(fca) is the index of fca
  lt: FloorCol # FloorCol -> Bool;
  % lt(fca,fcb) indicates if fca is less than fcb
var
  fca, fcb: FloorCol;
```

```
eqn
  index(c1) = 1;
  index(c2) = 2;
  index(c3) = 3;
  index(c4) = 4;
  index(c5) = 5;
  index(c6) = 6;
  index(c7) = 7;
  index(c8) = 8;
  index(c9) = 9;
  index(c10) = 10;
  lt(fca, fcb) = index(fca) < index(fcb);

% FloorPosPart operations
map
  index: FloorPosPart -> Pos;
  % index(ffpa) is the index of fppa
  lt: FloorPosPart # FloorPosPart -> Bool;
  % lt(fppa,fppb) indicates if fppa is less than fppb
var
  fppa, fppb: FloorPosPart;
eqn
  index(pa) = 1;
  index(pb) = 2;
  lt(fppa, fppb) = index(fppa) < index(fppb);

% FloorPos operations
map
  row: FloorPos -> FloorRow;
  % row(fpa) is the row of fpa
  col: FloorPos -> FloorCol;
  % col(fpa) is the column of fpa
  part: FloorPos -> FloorPosPart;
  % part(fpa) is the part of fpa
  lt: FloorPos # FloorPos -> Bool;
  % lt(fpa, fpb) indicates if fpa is less than fpb
  lt_r: FloorPos # FloorPos -> Bool;
  % lt_r(fpa, fpb) indicates if fpa is less than fpb, provided
  % row(fpa) = row(fpb) (auxiliary function needed by lt for efficiency)
  lt_rc: FloorPos # FloorPos -> Bool;
  % lt_rc(fpa, fpb) indicates if fpa is less than fpb, provided
  % row(fpa) = row(fpb) and col(fpa) = col(fpb)
  % (auxiliary function needed by lt_r for efficiency)
  gt: FloorPos # FloorPos -> Bool;
  % gt(fpa, fpb) indicates if fpa is greater than fpb
var
  fca: FloorCol;
  fppa: FloorPosPart;
  fpa, fpb: FloorPos;
eqn
  row(pos_r1(fca, fppa)) = r1;
  row(pos_r2(fca)) = r2;
  row(pos_r3(fca)) = r3;
  col(pos_r1(fca, fppa)) = fca;
  col(pos_r2(fca)) = fca;
  col(pos_r3(fca)) = fca;
  part(pos_r1(fca, fppa)) = fppa;
  row(fpa) == row(fpb) ->
    lt(fpa, fpb) = lt_r(fpa, fpb);
  row(fpa) != row(fpb) ->
    lt(fpa, fpb) = lt(row(fpa), row(fpb));
  col(fpa) == col(fpb) ->
    lt_r(fpa, fpb) = lt_rc(fpa, fpb);
  col(fpa) != col(fpb) ->
    lt_r(fpa, fpb) = lt(col(fpa), col(fpb));
  lt_rc(fpa, fpb) = row(fpa) == r1 && lt(part(fpa), part(fpb));
  gt(fpa,fpb) = lt(fpb, fpa);

% FloorPosList operations
map
  contains: FloorPos # FloorPosList -> Bool;
  % contains(fpa, fps) indicates if fps contains fpa
  overlap: FloorPosList # FloorPosList -> Bool;
  % overlap(fps, fpt) indicates if the elements of fps and fpt overlap
```

```
  rev: FloorPosList -> FloorPosList;
  % rev(fps) is the reverse of list fps
  grev: FloorPosList # FloorPosList -> FloorPosList;
  % grev(fps, fpt) is the reverse of fps concatenated with fpt
  % (auxiliary function needed for efficiency by rev)
var
  fpa, fpb: FloorPos;
  fps, fpt: FloorPosList;
eqn
  contains(fpa, []) = false;
  contains(fpa, fpa |> fps) = true;
  fpa != fpb -> contains(fpa, fpb |> fps) = contains(fpa, fps);
  overlap([], fpt)         = false;
  overlap(fpa |> fps, fpt) = contains(fpa, fpt) || overlap(fps, fpt);
  rev(fps)                 = grev(fps, []);
  grev([], fpt)            = fpt;
  grev(fpa |> fps, fpt)    = grev(fps, fpa |> fpt);

% ShuttlePos operations
map
  index: ShuttlePos -> Pos;
  % index(spa) is the index of spa
  row: ShuttlePos -> FloorRow;
  % row(spa) is the row of spa
  col: ShuttlePos -> FloorCol;
  % col(spa) is the column of spa
  lt: ShuttlePos # ShuttlePos -> Bool;
  % lt(spa, spb) indicates if spa is less than spb
  gt: ShuttlePos # ShuttlePos -> Bool;
  % gt(spa, spb) indicates if spa is greater than spb
  positions: ShuttlePos -> FloorPosList;
  % positions(spa) represents the floor positions corresponding to spa in
  % lowered position
  has_neighbour: ShuttlePos # DirRow -> Bool;
  % has_neighbour(spa, dr) indicates if spa has a successor in direction dr
  neighbour: ShuttlePos # DirRow -> ShuttlePos;
  % neighbour(spa, dr) is the neighbour of spa in directrion dr, provided it
  % exists
  connected: ShuttlePos # ShuttlePos -> Bool;
  % connected(spa, spb) indicates if the bottom of spa is connected to the top
  % of spb
var
  spa, spb: ShuttlePos;
eqn
  index(r1a) = 1;
  index(r2a) = 2;
  index(r3a) = 3;
  index(r1b) = 4;
  index(r2b) = 5;
  index(r3b) = 6;
  row(r1a) = r1;
  row(r2a) = r2;
  row(r3a) = r3;
  row(r1b) = r1;
  row(r2b) = r2;
  row(r3b) = r3;
  col(r1a) = c1;
  col(r2a) = c1;
  col(r3a) = c1;
  col(r1b) = c10;
  col(r2b) = c10;
  col(r3b) = c10;
  lt(spa, spb) = index(spa) < index(spb);
  gt(spa, spb) = index(spa) > index(spb);
  positions(r1a) = positions(b_r1a_sh);
  positions(r2a) = positions(b_r2a_sh);
  positions(r3a) = positions(b_r3a_sh);
  positions(r1b) = positions(b_r1b_sh);
  positions(r2b) = positions(b_r2b_sh);
  positions(r3b) = positions(b_r3b_sh);
  has_neighbour(spa, row_inc) = row(spa) != r3;
  has_neighbour(spa, row_dec) = row(spa) != r1;
  neighbour(r1a, row_inc) = r2a;
  neighbour(r2a, row_inc) = r3a;
```

```
  neighbour(r1b, row_inc) = r2b;
  neighbour(r2b, row_inc) = r3b;
  neighbour(r2a, row_dec) = r1a;
  neighbour(r3a, row_dec) = r2a;
  neighbour(r2b, row_dec) = r1b;
  neighbour(r3b, row_dec) = r2b;
  connected(spa, spb) =
    succ(index(spa)) == index(spb) &&
    col(spa) == col(spb);

% ShuttlePosSet operations
map
  is_set_sps: ShuttlePosSet -> Bool;
  % is_set_sps(sps) indicates if sps is a set
  % In the remainder of the data equation section, for every ShuttlePosSet sps
  % is_set_sps(sps) is a precondition.
  contains: ShuttlePos # ShuttlePosSet -> Bool;
  % contains(spa, sps) indicates if sps contains spa
  shuttles: FloorCol -> ShuttlePosSet;
  % shuttles(fc) represents the shuttle positions corresponding to fc
  positions_sps: ShuttlePosSet -> FloorPosList;
  % positions_sps(sps) represents the floor positions corresponding to sps
  connected_sps: ShuttlePosSet -> Bool;
  % connected_sps(sps) indicates if all elements of sps are connected
  has_neighbour: ShuttlePosSet # DirRow -> Bool;
  % has_neighbour(sps, dr) indicates is sps has a neighbour in direction dr
  neighbour_sps: ShuttlePosSet # DirRow -> ShuttlePos;
  % neighbour_sps(sps, dr) is the neighbour of sps in direction dr, provided it
  % exists
  add_neighbour: ShuttlePosSet # DirRow -> ShuttlePosSet;
  % add_neighbour(sps, dr) represents the sps extended with its neighbour in
  % direction dr, provided it exists
var
  spa, spb: ShuttlePos;
  sps: ShuttlePosSet;
  fc: FloorCol;
eqn
  is_set_sps([]) = true;
  is_set_sps(spa |> []) = true;
  is_set_sps(spa |> spb |> sps) =
    lt(spa, spb) && is_set_sps(spb |> sps);
  contains(spa, []) = false;
  contains(spa, spa |> sps) = true;
  spa != spb -> contains(spa, spb |> sps) = contains(spa, sps);
  shuttles(fc) = if(fc == c1, [r1a,r2a,r3a], if(fc == c10, [r1b,r2b,r3b], []));
  positions_sps([]) = [];
  positions_sps(spa |> sps) = positions(spa) ++ positions_sps(sps);
  connected_sps([])    = true;
  connected_sps([spa]) = true;
  connected_sps(spa |> spb |> sps) =
    connected(spa, spb) && connected_sps(spb |> sps);
  has_neighbour(sps, row_inc) = has_neighbour(rhead(sps), row_inc);
  has_neighbour(sps, row_dec) = has_neighbour(head(sps), row_dec);
  neighbour_sps(sps, row_inc) = neighbour(rhead(sps), row_inc);
  neighbour_sps(sps, row_dec) = neighbour(head(sps), row_dec);
  add_neighbour(sps, row_inc) = sps <| neighbour_sps(sps, row_inc);
  add_neighbour(sps, row_dec) = neighbour_sps(sps, row_dec) |> sps;

% ShuttleOrientation operations
map
  index: ShuttleOrientation -> Pos;
  % index(soa) is the index of soa
  lt: ShuttleOrientation # ShuttleOrientation -> Bool;
  % lt(soa, sob) indicates if soa is less than sob
  gt: ShuttleOrientation # ShuttleOrientation -> Bool;
  % gt(soa, sob) indicates if soa is greater than sob
  not: ShuttleOrientation -> ShuttleOrientation;
  % not(soa) indicates the opposite of soa
var
  soa, sob: ShuttleOrientation;
eqn
  index(lowered) = 1;
  index(tilted) = 2;
  lt(soa, sob) = index(soa) < index(sob);
```

```
  gt(soa, sob) = index(soa) > index(sob);
  not(lowered) = tilted;
  not(tilted) = lowered;

% Belt operations
map
  index: R1Belt -> Pos;
  index: R2Belt -> Pos;
  index: R3Belt -> Pos;
  % index(ba) is the index of belt ba
  row: R1Belt -> FloorRow;
  row: R2Belt -> FloorRow;
  row: R3Belt -> FloorRow;
  % row(ba) is the row of belt ba
  lt: R1Belt # R1Belt -> Bool;
  lt: R2Belt # R2Belt -> Bool;
  lt: R3Belt # R3Belt -> Bool;
  % lt(ba, bb) indicates if belt ba is less than belt bb
  positions: R1Belt -> FloorPosList;
  positions: R2Belt -> FloorPosList;
  positions: R3Belt -> FloorPosList;
  % positions(ba) represents the positions corresponding to belt ba
  connected: R1Belt # R1Belt -> Bool;
  connected: R2Belt # R2Belt -> Bool;
  connected: R3Belt # R3Belt -> Bool;
  % connected(ba, bb) indicates if the right of ba is connected to the left
  % of bb
  available: R1Belt # ShuttleState # LiftState -> Bool;
  available: R2Belt # ShuttleState # LiftState -> Bool;
  available: R3Belt # ShuttleState # LiftState -> Bool;
  % available(ba, shs, ls) indicates if belt ba is available for states shs
  % and ls
var
  r1ba, r1bb: R1Belt;
  r2ba, r2bb: R2Belt;
  r3ba, r3bb: R3Belt;
  fs: FloorState;
  shs: ShuttleState;
  ls: LiftState;
  ms: MoveSize;
eqn
  index(b_r1a_sh)  = 1;
  index(b_r1a)     = 2;
  index(b_r1lift)  = 3;
  index(b_r1b)     = 4;
  index(b_r1b_sh)  = 5;
  index(b_r2a_sh)  = 1;
  index(b_r2)      = 2;
  index(b_r2b_sh)  = 3;
  index(b_r3a_sh)  = 1;
  index(b_r3)      = 2;
  index(b_r3b_sh)  = 3;
  row(r1ba)        = r1;
  row(r2ba)        = r2;
  row(r3ba)        = r3;
  lt(r1ba, r1bb)   = index(r1ba) < index(r1bb);
  lt(r2ba, r2bb)   = index(r2ba) < index(r2bb);
  lt(r3ba, r3bb)   = index(r3ba) < index(r3bb);
  positions(b_r1a_sh) = [pos_r1(c1, pa), pos_r1(c1, pb)];
  positions(b_r1a)    = [pos_r1(c2, pa), pos_r1(c2, pb),
                         pos_r1(c3, pa), pos_r1(c3, pb),
                         pos_r1(c4, pa), pos_r1(c4, pb),
                         pos_r1(c5, pa), pos_r1(c5, pb),
                         pos_r1(c6, pa)];
  positions(b_r1lift) = [pos_r1(c6, pb), pos_r1(c7, pa)];
  positions(b_r1b)    = [pos_r1(c7, pb),
                         pos_r1(c8, pa), pos_r1(c8, pb),
                         pos_r1(c9, pa), pos_r1(c9, pb)];
  positions(b_r1b_sh) = [pos_r1(c10, pa), pos_r1(c10, pb)];
  positions(b_r2a_sh) = [pos_r2(c1)];
  positions(b_r2)     = [pos_r2(c2),
                         pos_r2(c3),
                         pos_r2(c4),
                         pos_r2(c5),
```

```
                        pos_r2(c6),
                        pos_r2(c7),
                        pos_r2(c8),
                        pos_r2(c9)];
  positions(b_r2b_sh) = [pos_r2(c10)];
  positions(b_r3a_sh) = [pos_r3(c1)];
  positions(b_r3)     = [pos_r3(c2),
                        pos_r3(c3),
                        pos_r3(c4),
                        pos_r3(c5),
                        pos_r3(c6),
                        pos_r3(c7),
                        pos_r3(c8),
                        pos_r3(c9)];
  positions(b_r3b_sh) = [pos_r3(c10)];
  available(b_r1a_sh, shs, ls) = shs(r1a, lowered) == avail;
  available(b_r1a, shs, ls) = true;
  available(b_r1lift, shs, ls) = ls == ls_basement;
  available(b_r1b, shs, ls) = true;
  available(b_r1b_sh, shs, ls) = shs(r1b, lowered) == avail;
  available(b_r2a_sh, shs, ls) = shs(r2a, lowered) == avail;
  available(b_r2, shs, ls) = true;
  available(b_r2b_sh, shs, ls) = shs(r2b, lowered) == avail;
  available(b_r3a_sh, shs, ls) = shs(r3a, lowered) == avail;
  available(b_r3, shs, ls) = true;
  available(b_r3b_sh, shs, ls) = shs(r3b, lowered) == avail;

% BeltSet operations
map
  is_set_b_r1: R1BeltSet -> Bool;
  is_set_b_r2: R2BeltSet -> Bool;
  is_set_b_r3: R3BeltSet -> Bool;
  % is_set_b_rX(bs) indicates if bs is a set
  % In the remainder of this data equations section each belt set bs has
  % is_set_b_rX(bs) as a precondition.
  contains: R1Belt # R1BeltSet -> Bool;
  contains: R2Belt # R2BeltSet -> Bool;
  contains: R3Belt # R3BeltSet -> Bool;
  % contains(ba, bs) indicates if ba is an element of bs
  positions_b_r1: R1BeltSet -> FloorPosList;
  positions_b_r2: R2BeltSet -> FloorPosList;
  positions_b_r3: R3BeltSet -> FloorPosList;
  % positions_b_rX(bs) represents the floor positions corresponding to the
  % belts in bs
  connected_b_r1: R1BeltSet -> Bool;
  connected_b_r2: R2BeltSet -> Bool;
  connected_b_r3: R3BeltSet -> Bool;
  % connected(bs) indicates if the belts in bs are connected
  available_b_r1: R1BeltSet # ShuttleState # LiftState -> Bool;
  available_b_r2: R2BeltSet # ShuttleState # LiftState -> Bool;
  available_b_r3: R3BeltSet # ShuttleState # LiftState -> Bool;
  % available_b_rX(bs, shs, ls) indicates if all belts in bs are available for
  % states shs and ls
var
  r1ba, r1bb: R1Belt;
  r2ba, r2bb: R2Belt;
  r3ba, r3bb: R3Belt;
  r1bs: R1BeltSet;
  r2bs: R2BeltSet;
  r3bs: R3BeltSet;
  fs: FloorState;
  shs: ShuttleState;
  ls: LiftState;
eqn
  is_set_b_r1([])             = true;
  is_set_b_r1(r1ba |> [])     = true;
  is_set_b_r1(r1ba |> r1bb |> r1bs) =
    lt(r1ba, r1bb) && is_set_b_r1(r1bb |> r1bs);
  is_set_b_r2([])             = true;
  is_set_b_r2(r2ba |> [])     = true;
  is_set_b_r2(r2ba |> r2bb |> r2bs) =
    lt(r2ba, r2bb) && is_set_b_r2(r2bb |> r2bs);
  is_set_b_r3([])             = true;
  is_set_b_r3(r3ba |> [])     = true;
```

```
  is_set_b_r3(r3ba |> r3bb |> r3bs) =
    lt(r3ba, r3bb) && is_set_b_r3(r3bb |> r3bs);
  contains(r1ba, []) = false;
  contains(r1ba, r1ba |> r1bs) = true;
  r1ba != r1bb -> contains(r1ba, r1bb |> r1bs) = contains(r1ba, r1bs);
  contains(r2ba, []) = false;
  contains(r2ba, r2ba |> r2bs) = true;
  r2ba != r2bb -> contains(r2ba, r2bb |> r2bs) = contains(r2ba, r2bs);
  contains(r3ba, []) = false;
  contains(r3ba, r3ba |> r3bs) = true;
  r3ba != r3bb -> contains(r3ba, r3bb |> r3bs) = contains(r3ba, r3bs);
  positions_b_r1([]) = [];
  positions_b_r1(r1ba |> r1bs) = positions(r1ba) ++ positions_b_r1(r1bs);
  positions_b_r2([]) = [];
  positions_b_r2(r2ba |> r2bs) = positions(r2ba) ++ positions_b_r2(r2bs);
  positions_b_r3([]) = [];
  positions_b_r3(r3ba |> r3bs) = positions(r3ba) ++ positions_b_r3(r3bs);
  connected_b_r1([]) = true;
  connected_b_r1(r1ba |> []) = true;
  connected_b_r1(r1ba |> r1bb |> r1bs) =
    connected(r1ba, r1bb) && connected_b_r1(r1bb |> r1bs);
  connected_b_r2([]) = true;
  connected_b_r2(r2ba |> []) = true;
  connected_b_r2(r2ba |> r2bb |> r2bs) =
    connected(r2ba, r2bb) && connected_b_r2(r2bb |> r2bs);
  connected_b_r3([]) = true;
  connected_b_r3(r3ba |> []) = true;
  connected_b_r3(r3ba |> r3bb |> r3bs) =
     connected(r3ba, r3bb) && connected_b_r3(r3bb |> r3bs);
  connected(r1ba, r1bb) = succ(index(r1ba)) == index(r1bb);
  connected(r2ba, r2bb) = succ(index(r2ba)) == index(r2bb);
  connected(r3ba, r3bb) = succ(index(r3ba)) == index(r3bb);
  available_b_r1([], shs, ls) = true;
  available_b_r1(r1ba |> r1bs, shs, ls) =
    available(r1ba, shs, ls) && available_b_r1(r1bs, shs, ls);
  available_b_r2([], shs, ls) = true;
  available_b_r2(r2ba |> r2bs, shs, ls) =
    available(r2ba, shs, ls) && available_b_r2(r2bs, shs, ls);
  available_b_r3([], shs, ls) = true;
  available_b_r3(r3ba |> r3bs, shs, ls) =
    available(r3ba, shs, ls) && available_b_r3(r3bs, shs, ls);

% Area operations
map
  overlap: Area # Area -> Bool;
  % area(a, b) indicates if area a and b overlap
var
  fps, fpt: FloorPosList;
  tc1a, tc1b, tc10a, tc10b: Bool;
eqn
  overlap(area(fps, tc1a, tc1b), area(fpt, tc10a, tc10b)) =
    overlap(fps, fpt) || (tc1a && tc1b) || (tc10a && tc10b);

% Instruction operations
map
  index: Instruction -> Pos;
  % index(i) is the index of instruction i
  lt: Instruction # Instruction -> Bool;
  % lt(i, j) indicates if i is less than j
  valid: Instruction -> Bool;
  % valid(i) indicates if instruction i is valid
  area: Instruction -> Area;
  % area(i) represents the area on which instruction i has any effect
  overlap: Instruction # Instruction -> Bool;
  % overlap(i, j) indicates if instruction i and j overlap
var
  r1bs: R1BeltSet;
  r2bs: R2BeltSet;
  r3bs: R3BeltSet;
  dc: DirCol;
  ms: MoveSize;
  sp: ShuttlePos;
  sps: ShuttlePosSet;
  so: ShuttleOrientation;
```

```
  dr: DirRow;
  lh: LiftHeight;
  i,j: Instruction;
eqn
  index(move_belts(r1bs, dc, ms)) = 1;
  index(move_belts(r2bs, dc, ms)) = 2;
  index(move_belts(r3bs, dc, ms)) = 3;
  index(move_shuttles(sps, so, dr)) = 4;
  index(tilt_shuttle(sp, so)) = 5;
  index(move_lift(lh)) = 6;
  index(rotate_lift) = 7;
  lt(i, j) = index(i) < index(j);
  valid(move_belts(r1bs, dc, ms)) = is_set_b_r1(r1bs);
  valid(move_belts(r2bs, dc, ms)) = is_set_b_r2(r2bs);
  valid(move_belts(r3bs, dc, ms)) = is_set_b_r3(r3bs);
  valid(move_shuttles(sps, so, dr)) = is_set_sps(sps);
  valid(tilt_shuttle(sp, so)) = true;
  valid(move_lift(lh)) = true;
  valid(rotate_lift) = true;
  area(move_belts(r1bs, dc, ms)) = area(positions_b_r1(r1bs), false, false);
  area(move_belts(r2bs, dc, ms)) = area(positions_b_r2(r2bs), false, false);
  area(move_belts(r3bs, dc, ms)) = area(positions_b_r3(r3bs), false, false);
  area(move_shuttles(sps, lowered, dr)) =
    area(positions_sps(add_neighbour(sps, dr)), false, false);
  area(move_shuttles(sps, tilted, dr)) =
    area([], col(head(sps)) == c1, col(head(sps)) == c10);
  area(tilt_shuttle(sp, so)) =
    area(positions(sp), col(sp) == c1, col(sp) == c10);
  area(move_lift(lh)) = area(positions(b_r1lift), false, false);
  area(rotate_lift) = area(positions_b_r1([b_r1a, b_r1lift, b_r1b]), false, false);
  overlap(i, j) = overlap(area(i), area(j));

% InstructionSet operations
map
  is_set_is: InstructionSet -> Bool;
  % is_set_is(is) indicates if is is a set
  % In the remainder of the data equation section, for every InstructionSet is
  % is_set_is(is) is a precondition.
  valid: InstructionSet -> Bool;
  % valid(is) indicates if the instructions in is are valid
  overlap: Instruction # InstructionSet -> Bool;
  % overlap(i, is) indicates if instruction i overlaps with any of the
  % instructions in is
var
  i,j: Instruction;
  is: InstructionSet;
eqn
  is_set_is([]) = true;
  is_set_is(i |> []) = true;
  is_set_is(i |> j |> is) = lt(i, j) && is_set_is(j |> is);
  valid([]) = true;
  valid(i |> is) = valid(i) && valid(is);
  overlap(i, []) = false;
  overlap(i, j |> is) = overlap(i, j) || overlap(i, is);

% AvailState operations
map
  not: AvailState -> AvailState;
  % not(asa) indicates the opposite of asa
eqn
  not(avail) = n_avail;
  not(n_avail) = avail;

% FloorState operations
map
  init_fs: FloorState;
  % init_fs is the initial floor state
  update: FloorPos # OccState # FloorState -> FloorState;
  cond_upd: FloorPos # OccState # FloorState -> FloorState;
  ins_upd: FloorPos # OccState # FloorState -> FloorState;
  % update(fpa, osa, fsa) represents fsa, where position fpa has value osa;
  % cond_upd and ins_upd are auxiliary functions with the same meaning that
  % are needed for efficiency
  ins_upd_fps: FloorPosList # OccState # FloorState -> FloorState;
```

```
  % ins_upd_fps(fps, osa, fsa) represents fsa, where positions fps have value
  % osa
  free: FloorPosList # FloorState -> Bool;
  % free(fps, fsa) indicates that all positions in fps are free for state fsa
  occupied: FloorPosList # FloorState -> Bool;
  % occupied(fps, fsa) indicates that all positions in fps are occupied for
  % state fsa
  end_free: FloorPosList # DirCol # Bool # FloorState -> Bool;
  % end_free(fps, dc, obl, fsa) indicates if the end of the list fps in
  % direction dc is free for state fsa; if obl then the one but last element
  % also needs to be free
  even_occ: FloorPosList # FloorState -> Bool;
  % even_occ(fps, fsa) indicates if the number of occupied positions in fps for
  % state fsa is even
  no_half_car: ShuttlePos # FloorState -> Bool;
  % no_half_car(spa, fsa) indicates if there is a half car positioned on spa
  % for state fsa
  no_half_car_sps: ShuttlePosSet # FloorState -> Bool;
  % no_half_car(sps, fsa) indicates if there is a half car positioned on any
  % of the elements of sps for state fsa
  shift_inc: FloorPosList # FloorState -> FloorState;
  % shift_inc(fps, fsa) represents fsa, where the states of the positions in
  % fps are shifted one position to the right; the first element in fps
  % gets state free
  gshift_inc: FloorPosList # FloorState # OccState -> FloorState;
  % gshift_inc(fps, fsa, osa) has the same meaning as shift_inc(fps, fsa), with
  % the exception that the first element in fps gets state osa
  % (auxiliary function needed by shift_inc)
  shift_dec: FloorPosList # FloorState -> FloorState;
  % shift_dec(fps, fsa) represents fsa, where the states of the positions in
  % fps are shifted one position to the left; the last element in fps gets
  % state free
  shift_inc_ctw: Bool # FloorPosList # FloorState -> FloorState;
  % shift_inc_ctw(ctw, fps, fsa) represents fsa where the states of the
  % positions in fps are shifted one position to the right;
  % if ctw then two positions are shifted
  shift_dec_ctw: Bool # FloorPosList # FloorState -> FloorState;
  % shift_dec_ctw(ctw, fps, fsa) represents fsa where the states of the
  % positions in fps are shifted one position to the left;
  % if ctw then two positions are shifted
  shift_inc_sps: ShuttlePosSet # FloorState -> FloorState;
  % shift_inc(sps, fsa) represents fsa, where the states of the shuttles in
  % sps are shifted one position downwards; the first element in sps
  % gets state free
  shift_dec_sps: ShuttlePosSet # FloorState -> FloorState;
  % shift_dec(sps, fsa) represents fsa, where the states of the shuttles in
  % sps are shifted one position upwards; the last element in sps gets state
  % free
var
  b: Bool;
  fsa, fsb: FloorState;
  fpa, fpb: FloorPos;
  osa, osb: OccState;
  fps: FloorPosList;
  spa: ShuttlePos;
  sps: ShuttlePosSet;
eqn
  init_fs(fpa) = free;
  fpa == fpb  -> update(fpa, osa, fsa)(fpb) = osa;
  fpa != fpb  -> update(fpa, osa, fsa)(fpb) = fsa(fpb);
  cond_upd(fpa, osa, fsa) =
    if(fsa(fpa) == osa, fsa, update(fpa, osa, fsa));
  ins_upd(fpa, osa, init_fs) = cond_upd(fpa, osa, init_fs);
  lt(fpa,fpb) ->
    ins_upd(fpa, osa, update(fpb, osb, fsa)) =
    cond_upd(fpa, osa, update(fpb, osb, fsa));
  gt(fpa,fpb) ->
    ins_upd(fpa, osa, update(fpb, osb, fsa)) =
    update(fpb, osb, ins_upd(fpa, osa, fsa));
  fpa == fpb  ->
    ins_upd(fpa, osa, update(fpb, osb, fsa)) =
    cond_upd(fpa, osa, fsa);
  ins_upd_fps([], osa, fsa) = fsa;
  ins_upd_fps(fpa |> fps, osa, fsa) =
```

```
    ins_upd(fpa, osa, ins_upd_fps(fps, osa, fsa));
  free([], fsa) = true;
  free(fpa |> fps, fsa) = fsa(fpa) == free && free(fps, fsa);
  occupied(fpa |> fps, fsa) = fsa(fpa) == occupied && occupied(fps, fsa);
  occupied([], fsa) = true;
  end_free(fps, col_inc, false, fsa) =
    fsa(head(rev(fps))) == free;
  end_free(fps, col_inc, true, fsa) =
    fsa(head(l)) == free && fsa(head(tail(l))) == free
      whr l = rev(fps) end;
  end_free(fps, col_dec, false, fsa) =
    fsa(head(fps)) == free;
  end_free(fps, col_dec, true, fsa) =
    fsa(head(fps)) == free && fsa(head(tail(fps))) == free;
  even_occ([], fsa) = true;
  fsa(fpa) == free ->
    even_occ(fpa |> fps, fsa) = even_occ(fps, fsa);
  fsa(fpa) == occupied ->
    even_occ(fpa |> fps, fsa) = !even_occ(fps, fsa);
  row(spa) != r1 ->
    no_half_car(spa, fsa) = true;
  row(spa) == r1 ->
    no_half_car(spa, fsa) = even_occ(positions(spa), fsa);
  no_half_car_sps([], fsa) = true;
  no_half_car_sps(spa |> sps, fsa) =
    no_half_car(spa, fsa) && no_half_car_sps(sps, fsa);
  shift_inc(fps, fsa) = gshift_inc(fps, fsa, free);
  gshift_inc([], fsa, osa) = fsa;
  gshift_inc([fpa], fsa, osa) = ins_upd(fpa, osa, fsa);
  gshift_inc(fpa |> fps, fsa, osa) =
    ins_upd(fpa, osa, gshift_inc(fps, fsa, fsa(fpa)));
  shift_dec([], fsa) = fsa;
  shift_dec([fpa], fsa) = ins_upd(fpa, free, fsa);
  shift_dec(fpa |> fpb |> fps, fsa) =
    ins_upd(fpa, fsa(fpb), shift_dec(fpb |> fps, fsa));
  shift_inc_ctw(false, fps, fsa) =
    shift_inc(fps, fsa);
  shift_inc_ctw(true, fps, fsa) =
    shift_inc(fps, shift_inc(fps, fsa));
  shift_dec_ctw(false, fps, fsa) =
    shift_dec(fps, fsa);
  shift_dec_ctw(true, fps, fsa) =
    shift_dec(fps, shift_dec(fps, fsa));
  shift_inc_sps([], fsa) = fsa;
  row(spa) == r1 ->
    shift_inc_sps(spa |> sps, fsa) =
      ins_upd_fps(positions(spa), free,
        shift_inc(positions_sps(spa |> sps), fsa));
  row(spa) != r1 ->
    shift_inc_sps(spa |> sps, fsa) = shift_inc(positions_sps(spa |> sps), fsa);
  shift_dec_sps([], fsa) = fsa;
  row(spa) == r1 ->
    shift_dec_sps(spa |> sps, fsa) =
      ins_upd_fps(positions(spa), fsa(head(positions(head(sps)))),
        shift_dec(positions_sps(spa |> sps), fsa));
  row(spa) != r1 ->
    shift_dec_sps(spa |> sps, fsa) = shift_dec(positions_sps(spa |> sps), fsa);

% ShuttleState operations
map
  init_shs: ShuttleState;
  % initial shuttle state
  update: ShuttlePos # ShuttleOrientation # AvailState # ShuttleState ->
    ShuttleState;
  cond_upd: ShuttlePos # ShuttleOrientation # AvailState # ShuttleState ->
    ShuttleState;
  ins_upd: ShuttlePos # ShuttleOrientation # AvailState # ShuttleState ->
    ShuttleState;
  % update(spa, soa, asa, fsa) represents fsa, where position fpa in
  % orientation soa has value asa; cond_upd and ins_upd are auxiliary functions
  % with the same meaning that are needed for efficiency
  available: ShuttlePos # ShuttleOrientation # ShuttleState -> Bool;
  % available(spa, soa, shs) indicates if shuttle spa is available in
  % orientation soa for state shs
```

```
  available_sps: ShuttlePosSet # ShuttleOrientation # ShuttleState -> Bool;
  % available(sps, soa, shs) indicates if all shuttles in sps are available in
  % orientation soa for state shs
  shift_inc: ShuttlePosSet # ShuttleOrientation # ShuttleState -> ShuttleState;
  % shift_inc(sps, soa, shs) represents shs, where the states of the shuttles
  % in sps are shifted one shuttle downwards in orientation soa; the first
  % element in sps gets state unavailable
  gshift_inc: ShuttlePosSet # ShuttleOrientation # ShuttleState # AvailState
    -> ShuttleState;
  % gshift_inc(sps, soa, shs, asa) has the same meaning as shift_inc(sps, soa,
  % shs), with the exception that the first element in sps gets state asa
  % (auxiliary function needed by shift_inc)
  shift_dec: ShuttlePosSet # ShuttleOrientation # ShuttleState -> ShuttleState;
  % shift_dec(sps, soa, shs) represents shs, where the states of the shuttles
  % in sps are shifted one shuttle upwards in orientation soa; the last
  % element in sps gets state unavailable
var
  b: Bool;
  shs, shsa: ShuttleState;
  spa, spb: ShuttlePos;
  sps: ShuttlePosSet;
  soa, sob: ShuttleOrientation;
  asa, asb: AvailState;
eqn
  init_shs(spa, lowered) = if(row(spa) != r3, avail, n_avail);
  init_shs(spa, tilted) = if(row(spa) != r3, n_avail, avail);
  spa == spb  && soa == sob  ->
    update(spa, soa, asa, shs)(spb, sob) = asa;
  spa != spb  || soa != sob  ->
    update(spa, soa, asa, shs)(spb, sob) = shs(spb, sob);
  cond_upd(spa, soa, asa, shs) =
    if(shs(spa, soa) == asa, shs, update(spa, soa, asa, shs));
  ins_upd(spa, soa, asa, init_shs) =
    cond_upd(spa, soa, asa, init_shs);
  lt(spa,spb) || (spa == spb && lt(soa, sob)) ->
    ins_upd(spa, soa, asa, update(spb, sob, asb, shs)) =
    cond_upd(spa, soa, asa, update(spb, sob, asb, shs));
  gt(spa,spb) || (spa == spb && gt(soa, sob)) ->
    ins_upd(spa, soa, asa, update(spb, sob, asb, shs)) =
    update(spb, sob, asb, ins_upd(spa, soa, asa, shs));
  spa == spb && soa == sob ->
    ins_upd(spa, soa, asa, update(spb, sob, asb, shs)) =
    cond_upd(spa, soa, asa, shs);
  available(spa, soa, shs) = shs(spa, soa) == avail;
  available_sps([], soa, shs) = true;
  available_sps(spa |> sps, soa, shs) =
    available(spa, soa, shs) && available_sps(sps, soa, shs);
  shift_inc(sps, soa, shs) = gshift_inc(sps, soa, shs, n_avail);
  gshift_inc([], soa, shs, asa)    = shs;
  gshift_inc([spa], soa, shs, asa) =
    ins_upd(spa, soa, asa, shs);
  gshift_inc(spa |> sps, soa, shs, asa) =
    ins_upd(spa, soa, asa, gshift_inc(sps, soa, shs, shs(spa, soa)));
  shift_dec([], soa, shs)       = shs;
  shift_dec([spa], soa, shs)    = ins_upd(spa, soa, n_avail, shs);
  shift_dec(spa |> spb |> sps, soa, shs) =
    ins_upd(spa, soa, shs(spb, soa), shift_dec(spb |> sps, soa, shs));

% LiftState operations
map
  height: LiftState -> LiftHeight;
  % height(ls) represents the height of state ls
  occupied: LiftState # FloorState -> Bool;
  % occupied(ls, fs) indicates if the lift is occupieed for states ls and fs
  make_ls: LiftHeight # OccState -> LiftState;
  % make_ls(lh, os) represents the lift state corresponding to height lh and
  % occupied state os
var
  b: Bool;
  fsa: FloorState;
  osa: OccState;
eqn
  height(lsf_street)  = street;
  height(lso_street)  = street;
```

```
  height(lsf_rotate)  = rotate;
  height(lso_rotate)  = rotate;
  height(ls_basement) = basement;
  occupied(lsf_street, fsa) = false;
  occupied(lso_street, fsa) = true;
  occupied(lsf_rotate, fsa) = false;
  occupied(lso_rotate, fsa) = true;
  occupied(ls_basement, fsa) = occupied(positions(b_r1lift), fsa);
  make_ls(street, occupied) = lso_street;
  make_ls(street, free) = lsf_street;
  make_ls(rotate, occupied) = lso_rotate;
  make_ls(rotate, free) = lsf_rotate;
  make_ls(basement, osa) = ls_basement;

% GlobalState operations
map
  init_gs: GlobalState;
  % init_gs is the initial global state
  allowed: Instruction # GlobalState -> Bool;
  % allowed(i, gs) indicates if instruction i is allowed given global state gs
  nextstate: Instruction # ExecResult # GlobalState -> GlobalState;
  % nextstate(i, r, gs) represents the global state after execution of
  % instruction i with result r in state gs
  allowed: InstructionSet # GlobalState -> Bool;
  % allowed(is, gs) indicates if instruction set is is allowed given global
  % state gs
  nextstate: InstructionSet # ExecResult # GlobalState -> GlobalState;
  % nextstate(is, r, gs) represents the global state after execution of
  % instruction set is with result r in state gs
  possible: Event # GlobalState -> Bool;
  % possible(e, gs) indicates if event e is possible given global state gs
  nextstate: Event # GlobalState -> GlobalState;
  % nextstate(e, gs) represents the global state after the event e has occurred
  % in state gs
var
  gs, gsa: GlobalState;
  fs, fsa: FloorState;
  shs, shsa: ShuttleState;
  ls, lsa: LiftState;
  lh, lha: LiftHeight;
  p,q: FloorPos;
  s,t: OccState;
  i,j: Instruction;
  r1bs: R1BeltSet;
  r2bs: R2BeltSet;
  r3bs: R3BeltSet;
  dc: DirCol;
  ms: MoveSize;
  sps: ShuttlePosSet;
  dr: DirRow;
  sp, spa, spb: ShuttlePos;
  so: ShuttleOrientation;
  is: InstructionSet;
eqn
  init_gs =
    glob_state(
      init_fs,
      init_shs,
      lsf_street);
  allowed(move_belts(r1bs, dc, ms), glob_state(fs, shs, ls)) =
    r1bs != [] &&
    connected_b_r1(r1bs) &&
    available_b_r1(r1bs, shs, ls) &&
    end_free(l, dc, ms == full, fs) &&
    even_occ(l, fs)
    whr l = positions_b_r1(r1bs) end;
  allowed(move_belts(r2bs, dc, ms), glob_state(fs, shs, ls)) =
    ms == full &&
    r2bs != [] &&
    connected_b_r2(r2bs) &&
    available_b_r2(r2bs, shs, ls) &&
    end_free(positions_b_r2(r2bs), dc, false, fs);
  allowed(move_belts(r3bs, dc, ms), glob_state(fs, shs, ls)) =
    ms == full &&
```

```
  r3bs != [] &&
  connected_b_r3(r3bs) &&
  available_b_r3(r3bs, shs, ls) &&
  end_free(positions_b_r3(r3bs), dc, false, fs);
allowed(move_shuttles(sps, so, dr), glob_state(fs, shs, ls)) =
  sps != [] &&
  connected_sps(sps) &&
  available_sps(sps, so, shs) &&
  % placing a comment in front of the following conjunct allows the shuttles
  % to be moved even if a car is placed half on the shuttle
  (so == lowered => no_half_car_sps(sps, fs)) &&
  has_neighbour(sps, dr) &&
  !available(neighbour_sps(sps, dr), so, shs);
allowed(tilt_shuttle(sp, lowered), glob_state(fs, shs, ls)) =
  available_sps(shuttles(col(sp)), lowered, shs) &&
  free(positions(sp), fs);
allowed(tilt_shuttle(sp, tilted), glob_state(fs, shs, ls)) =
  shs(sp, tilted) == avail &&
  shs(sp, lowered) == n_avail;
ls == ls_basement ->
  allowed(move_lift(lh), glob_state(fs, shs, ls)) =
  lh != height(ls) &&
  % replacing the following conjunct by
  %   even_occ(positions(b_r1lift), fs);
  % allows the lift to move up if two cars are placed half on the lift belt
  if(free(positions(b_r1lift), fs), true,
    even_occ(positions_b_r1([b_r1a_sh, b_r1a]), fs) &&
    even_occ(positions_b_r1([b_r1b, b_r1b_sh]), fs));
ls != ls_basement ->
  allowed(move_lift(lh), glob_state(fs, shs, ls)) =
  lh != height(ls);
allowed(rotate_lift, glob_state(fs, shs, ls)) =
  height(ls) == rotate &&
  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);
nextstate(move_belts(r1bs, col_inc, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_inc_ctw(ms == full, positions_b_r1(r1bs), fs), shs, ls);
nextstate(move_belts(r1bs, col_dec, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_dec_ctw(ms == full, positions_b_r1(r1bs), fs), shs, ls);
nextstate(move_belts(r2bs, col_inc, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_inc(positions_b_r2(r2bs), fs), shs, ls);
nextstate(move_belts(r2bs, col_dec, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_dec(positions_b_r2(r2bs), fs), shs, ls);
nextstate(move_belts(r3bs, col_inc, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_inc(positions_b_r3(r3bs), fs), shs, ls);
nextstate(move_belts(r3bs, col_dec, ms), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_dec(positions_b_r3(r3bs), fs), shs, ls);
nextstate(move_shuttles(sps, lowered, row_inc), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_inc_sps(l, fs), shift_inc(l, lowered, shs), ls)
    whr l = add_neighbour(sps, row_inc) end;
nextstate(move_shuttles(sps, lowered, row_dec), ok, glob_state(fs, shs, ls)) =
  glob_state(shift_dec_sps(l, fs), shift_dec(l, lowered, shs), ls)
    whr l = add_neighbour(sps, row_dec) end;
nextstate(move_shuttles(sps, tilted, row_inc), ok, glob_state(fs, shs, ls)) =
  glob_state(fs, shift_inc(add_neighbour(sps, row_inc), tilted, shs), ls);
nextstate(move_shuttles(sps, tilted, row_dec), ok, glob_state(fs, shs, ls)) =
  glob_state(fs, shift_dec(add_neighbour(sps, row_dec), tilted, shs), ls);
nextstate(tilt_shuttle(sp, so), ok, glob_state(fs, shs, ls)) =
  glob_state(fs,
    ins_upd(sp, so, n_avail, ins_upd(sp, not(so), avail, shs)),
    ls);
occupied(ls, fs) ->
  nextstate(move_lift(lh), ok, glob_state(fs, shs, ls)) =
    glob_state(
      if(height(ls) != basement && lh == basement,
        ins_upd_fps(positions(b_r1lift), occupied, fs),
        if(height(ls) == basement && lh != basement,
          ins_upd_fps(positions(b_r1lift), free, fs),
          fs
          )
        ),
      shs, make_ls(lh, occupied));
!occupied(ls, fs) ->
  nextstate(move_lift(lh), ok, glob_state(fs, shs, ls)) =
```

```
    glob_state(fs, shs, make_ls(lh, free));
nextstate(rotate_lift, ok, gs) = gs;
nextstate(i, fail, gs) = gs;
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);
nextstate([], ok, gs) = gs;
nextstate(i |> is, ok, gs) = nextstate(i, ok, nextstate(is, ok, gs));
nextstate(is, fail, gs) = gs;
possible(add_car, glob_state(fs, shs, ls)) =
  ls == lsf_street;
possible(remove_car, glob_state(fs, shs, ls)) =
  ls == lso_street;
nextstate(add_car, glob_state(fs, shs, ls)) =
  glob_state(fs, shs, lso_street);
nextstate(remove_car, glob_state(fs, shs, ls)) =
  glob_state(fs, shs, lsf_street);
```

Appendix C: Process specifications

In the following sections, the formal specification of the process part of the safety layer is given, followed by the specifications after reducing complexity to enable simulation.

C.1 General specification

```
sort
  ProcState = struct ps_idle | ps_await_state | ps_ack_deny
                   | ps_req | ps_exec | ps_ack_exec;
  Layer = struct logical | safety | hardware;

act
  snd_req, rcv_req, com_req: Layer # Layer # InstructionSet;
  snd_ack_req, rcv_ack_req, com_ack_req: Layer # Layer # InstructionSet;
  snd_deny_req, rcv_deny_req, com_deny_req: Layer # Layer # InstructionSet;
  snd_ack_exec, rcv_ack_exec, ack_exec:Layer # Layer # InstructionSet # ExecResult;
  snd_state, rcv_state, com_state: Layer # Layer # GlobalState;
  snd_event, rcv_event, com_event: Layer # Layer # Event;

proc
  SL(ps: ProcState, gs_sl: GlobalState, is: InstructionSet, r: ExecResult) =
    (ps == ps_idle) ->
      (sum isa: InstructionSet. valid(isa) ->
        rcv_req(logical, safety, isa) .
          SL(ps_ack_deny, gs_sl, isa, r)
      )
  +
    (ps == ps_await_state) ->
      (sum gs: GlobalState. rcv_state(hardware, safety, gs).
        SL(ps_ack_deny, gs, is, 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) ->
      (sum ra: ExecResult.
        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) ->
      (sum 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);
```

C.2 Specification after reduction 1

In the following specification, instruction sets are replaced by single instructions.

```
sort
  ProcState = struct ps_idle | ps_await_state | ps_ack_deny
                   | ps_req | ps_exec | ps_ack_exec;
  Layer = struct logical | safety | hardware;

act
  snd_req, rcv_req, com_req: Layer # Layer # Instruction;
  snd_ack_req, rcv_ack_req, com_ack_req: Layer # Layer # Instruction;
  snd_deny_req, rcv_deny_req, com_deny_req: Layer # Layer # Instruction;
  snd_ack_exec, rcv_ack_exec, ack_exec:Layer # Layer # Instruction # ExecResult;
  snd_state, rcv_state, com_state: Layer # Layer # GlobalState;
  snd_event, rcv_event, com_event: Layer # Layer # Event;

proc
  SL(ps: ProcState, gs_sl: GlobalState, i: Instruction, r: ExecResult) =
    (ps == ps_idle) ->
      (sum j: Instruction. valid(j) ->
        rcv_req(logical, safety, j) .
          SL(ps_ack_deny, gs_sl, j, r)
      )
  +
    (ps == ps_await_state) ->
      (sum gs: GlobalState. rcv_state(hardware, safety, gs).
        SL(ps_ack_deny, gs, i, r)
      )
  +
    (ps == ps_ack_deny) ->
      (allowed(i, gs_sl) ->
        (snd_ack_req(safety, logical, i) . SL(ps_req, gs_sl, i, r)),
        (snd_deny_req(safety, logical, i). SL(ps_idle, gs_sl, i, r)))
  +
    (ps == ps_req) ->
      snd_req(safety, hardware, i) . SL(ps_exec, gs_sl, i, r)
  +
    (ps == ps_exec) ->
      (sum ra: ExecResult.
        rcv_ack_exec(safety, hardware, i, ra).
          SL(ps_ack_exec, nextstate(i, ra, gs_sl), i, ra)
      )
  +
    (ps == ps_ack_exec) ->
      snd_ack_exec(safety, logical, i, r) . SL(ps_idle, gs_sl, i, r)
  +
    (ps == ps_idle) ->
      (sum e: Event.
        possible(e, gs_sl) -> rcv_event(hardware, safety, e) .
          SL(ps, nextstate(e, gs_sl), i, r)
      )
  ;

init
  SL(ps_idle, init_gs, move_lift(street), ok);
```

C.3 Specification after reduction 2

The following specification abstracts from non-essential message passing:

```
act
  exec: Instruction;
  occur: Event;

proc
  S(gs_sl: GlobalState) =
    sum i: Instruction. (valid(i) && allowed(i, gs_sl)) ->
      exec(i) . S(nextstate(i, ok, gs_sl))
  +
    sum e: Event. possible(e, gs_sl) ->
      occur(e) . S(nextstate(e, gs_sl))
  ;

init
  S(init_gs);
```

Appendix D: Formal verification

For the formal verification of the requirements the specification is reduced by changing the data specification and adding behaviour to the process specification.

D.1 Specification after reduction 3

The specification after reduction 3 is the specification with the process specification of appendix C.3, together with the data specification of appendix B, in which the capacities of the belts *b_r1a*, *b_r1b*, *b_r2* and *b_r3* are reduced.

```
positions(b_r1a)    = [pos_r1(c2, pa), pos_r1(c2, pb), pos_r1(c6, pa)];
positions(b_r1b)    = [pos_r1(c7, pb), pos_r1(c9, pa), pos_r1(c9, pb)];
positions(b_r2)     = [pos_r2(c2), pos_r2(c9)];
positions(b_r3)     = [pos_r3(c2), pos_r3(c9)];
```

D.2 Specification after adding error actions

The requirements of section 6.1 are still rather high-level. In order to formalise them in mCRL2, we translate them to a more detailed level:

1. Belts:
   (a) For each belt *b*, belt set *bs* and move size *ms*, the instruction *move_belts(bs, col_inc, ms)* should not be allowed if:
      - The right-most position of belt *b* of size *ms* is occupied.
      - *b* is the right-most belt in *bs*.

      Likewise for the left-most position and belt for instruction *move_belts(bs, col_dec, ms)*.
   (b) For each shuttle *s*, belt set *bs*, direction *d* and move size *ms*, the instruction *move_belts(bs, d, ms)* should not be allowed if the following conditions hold:
      - Shuttle *s* is available in lowered position and contains (a part of) a car.
      - The belt corresponding to shuttle *s* is in the belt set *bs*.
      - Direction *d* is pointing to the wall.
      - Move size *ms* could move the car into the wall.
   (c) For each belt set *bs*, direction *d* and move size *ms*, the instruction *move_belts(bs, d, ms)* should not be allowed if *bs* contains a belt that is not available.
2. Shuttles:
   (a) For all sets of shuttle positions *sps*, orientations *o* and directions *d*, the instruction *move_shuttles(sps, o, d)* should not be allowed if it contains a shuttle that directly faces a wall in direction *d*.
   (b) For all shuttle positions *s*, sets of shuttle positions *sps* and directions *d*, the instruction *move_shuttles(sps, lowered, d)* should not be allowed if:
      - *s* is available in lowered position.
      - *s* is the neighbour of sps in direction *d*.
   (c) for all shuttles *s*, sets of shuttle positions *sps* and directions *d*, the instruction *move_shuttles(sps, lowered, d)* should not be allowed if:

- *s* is neither completely free nor completely occupied.
- *s* is an element of the set *sps*.

(d) For all shuttles *s* and orientations *o*, the instruction *tilt_shuttle(s, o)* should not be allowed if there is a car on the belt corresponding to *s*.

(e) For all shuttles *s* and orientations *o*, the instruction *tilt_shuttle(s, o)* should not be allowed if there is a shuttle in position *s* in both orientations.

3. Lift:

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

- The lift is at the basement level.
- The lift contains one or two cars that are placed half on the lift.

(b) The instruction *rotate_lift* should not be allowed if one of the following holds:

- The lift is not at rotate height.
- The three half positions belts at the left and the right of the lift are not completely free.

These low-level requirements are specified in mCRL2 as conditions of error actions. This is expressed by the following specification that is used for verification.

```
sort
  Requirement = struct req1a | req1b | req1c
                     | req2a | req2b | req2c | req2d | req2e
                     | req3a | req3b;
act
  exec: Instruction;
  occur: Event;
  error: Requirement # Pos;

proc
  V(gs_sl: GlobalState) =
  % normal behaviour
    sum i: Instruction. (valid(i) && allowed(i, gs_sl)) ->
      exec(i) . V(nextstate(i, ok, gs_sl))
  +
    sum e: Event. possible(e, gs_sl) ->
      occur(e) . V(nextstate(e, gs_sl))
  % added behaviour with the purpose of checking the requirements;
  % if this results in more behaviour, a requirement is not satisfied
  % Requirement 1a
  +
    sum i: Instruction, b: R1Belt, ms: MoveSize, bs: R1BeltSet.
      (!end_free(positions(b), col_inc, ms == full, fs(gs_sl)) &&
       rhead(bs) == b &&
       i == move_belts(bs, col_inc, ms) &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req1a, 1) . delta
  +
    sum i: Instruction, b: R1Belt, ms: MoveSize, bs: R1BeltSet.
      (!end_free(positions(b), col_dec, ms == full, fs(gs_sl)) &&
       head(bs) == b &&
       i == move_belts(bs, col_dec, ms) &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req1a, 2) . delta
  +
    sum i: Instruction, b: R2Belt, ms: MoveSize, bs: R2BeltSet.
      (!end_free(positions(b), col_inc, false, fs(gs_sl)) &&
       rhead(bs) == b &&
       i == move_belts(bs, col_inc, ms) &&
       valid(i) &&
       allowed(i, gs_sl)
```

```
    ) -> error(req1a, 3) . delta
+
  sum i: Instruction, b: R2Belt, ms: MoveSize, bs: R2BeltSet.
    (!end_free(positions(b), col_dec, false, fs(gs_sl)) &&
     head(bs) == b &&
     i == move_belts(bs, col_dec, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1a, 4) . delta
+
  sum i: Instruction, b: R3Belt, ms: MoveSize, bs: R3BeltSet.
    (!end_free(positions(b), col_inc, false, fs(gs_sl)) &&
     rhead(bs) == b &&
     i == move_belts(bs, col_inc, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1a, 5) . delta
+
  sum i: Instruction, b: R3Belt, ms: MoveSize, bs: R3BeltSet.
    (!end_free(positions(b), col_dec, false, fs(gs_sl)) &&
     head(bs) == b &&
     i == move_belts(bs, col_dec, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1a, 6) . delta
% Requirement 1b
+
  sum i: Instruction, ms: MoveSize, bs: R1BeltSet.
    (available(r1a, lowered, shs(gs_sl)) &&
     !free(positions(r1a), fs(gs_sl)) &&
     i == move_belts(bs, col_dec,
       if(!occupied(positions(r1a), fs(gs_sl)), full, ms)) &&
     contains(b_r1a_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 1) . delta
+
  sum i: Instruction, ms: MoveSize, bs: R1BeltSet.
    (available(r1b, lowered, shs(gs_sl)) &&
     !free(positions(r1b), fs(gs_sl)) &&
     i == move_belts(bs, col_inc,
       if(!occupied(positions(r1b), fs(gs_sl)), full, ms)) &&
     contains(b_r1b_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 2) . delta
+
  sum i: Instruction, ms: MoveSize, bs: R2BeltSet.
    (available(r2a, lowered, shs(gs_sl)) &&
     !free(positions(r2a), fs(gs_sl)) &&
     i == move_belts(bs, col_dec, ms) &&
     contains(b_r2a_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 3) . delta
+
  sum i: Instruction, ms: MoveSize, bs: R2BeltSet.
    (available(r2b, lowered, shs(gs_sl)) &&
     !free(positions(r2b), fs(gs_sl)) &&
     i == move_belts(bs, col_inc, ms) &&
     contains(b_r2b_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 4) . delta
+
  sum i: Instruction, ms: MoveSize, bs: R3BeltSet.
    (available(r3a, lowered, shs(gs_sl)) &&
     !free(positions(r3a), fs(gs_sl)) &&
     i == move_belts(bs, col_dec, ms) &&
     contains(b_r3a_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 5) . delta
+
```

```
  sum i: Instruction, ms: MoveSize, bs: R3BeltSet.
    (available(r3b, lowered, shs(gs_sl)) &&
     !free(positions(r3b), fs(gs_sl)) &&
     i == move_belts(bs, col_inc, ms) &&
     contains(b_r3b_sh, bs) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1b, 6) . delta
% Requirement 1c
+
  sum i: Instruction, dc: DirCol, ms: MoveSize, bs: R1BeltSet.
    (!available_b_r1(bs, shs(gs_sl), ls(gs_sl)) &&
     i == move_belts(bs, dc, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1c, 1) . delta
+
  sum i: Instruction, dc: DirCol, ms: MoveSize, bs: R2BeltSet.
    (!available_b_r2(bs, shs(gs_sl), ls(gs_sl)) &&
     i == move_belts(bs, dc, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1c, 2) . delta
+
  sum i: Instruction, dc: DirCol, ms: MoveSize, bs: R3BeltSet.
    (!available_b_r3(bs, shs(gs_sl), ls(gs_sl)) &&
     i == move_belts(bs, dc, ms) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req1c, 3) . delta
% Requirement 2a
+
  sum i: Instruction, sps: ShuttlePosSet, so: ShuttleOrientation.
    ((contains(r1a, sps) || contains(r1b, sps)) &&
     i == move_shuttles(sps, so, row_dec) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req2a, 1) . delta
+
  sum i: Instruction, sps: ShuttlePosSet, so: ShuttleOrientation.
    ((contains(r3a, sps) || contains(r3b, sps)) &&
     i == move_shuttles(sps, so, row_inc) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req2a, 2) . delta
% Requirement 2b
+
  sum i: Instruction, sp: ShuttlePos, sps: ShuttlePosSet.
    (available(sp, lowered, shs(gs_sl)) &&
     connected(rhead(sps), sp) &&
     i == move_shuttles(sps, lowered, row_inc) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req2b, 1) . delta
+
  sum i: Instruction, sp: ShuttlePos, sps: ShuttlePosSet.
    (available(sp, lowered, shs(gs_sl)) &&
     connected(sp, head(sps)) &&
     i == move_shuttles(sps, lowered, row_dec) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req2b, 2) . delta
% Requirement 2c
+
  sum i: Instruction, sp: ShuttlePos, sps: ShuttlePosSet, dr: DirRow.
    (!free(positions(sp),fs(gs_sl)) &&
     !occupied(positions(sp),fs(gs_sl)) &&
     contains(sp, sps) &&
     i == move_shuttles(sps, lowered, dr) &&
     valid(i) &&
     allowed(i, gs_sl)
    ) -> error(req2c, 1) . delta
% Requirement 2d
+
```

```
    sum i: Instruction, sp: ShuttlePos, so: ShuttleOrientation.
      (!free(positions(sp),fs(gs_sl)) &&
       i == tilt_shuttle(sp, so) &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req2d, 1) . delta
  % Requirement 2e
  +
    sum i: Instruction, sp: ShuttlePos, so: ShuttleOrientation.
      (available(sp, so, shs(gs_sl)) &&
       available(sp, not(so), shs(gs_sl)) &&
       i == tilt_shuttle(sp, so) &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req2e, 1) . delta
  % Requirement 3a
  +
    sum i: Instruction, lh: LiftHeight.
      (ls(gs_sl) == ls_basement &&
       !free(positions(b_r1lift), fs(gs_sl)) &&
       !(even_occ(positions_b_r1([b_r1a_sh, b_r1a]), fs(gs_sl)) &&
         even_occ(positions_b_r1([b_r1b, b_r1b_sh]), fs(gs_sl))) &&
       i == move_lift(lh) &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req3a, 1) . delta
  % Requirement 3b
  +
    sum i: Instruction.
      ((height(ls(gs_sl)) != rotate ||
        !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(gs_sl))
       ) &&
       i == rotate_lift &&
       valid(i) &&
       allowed(i, gs_sl)
      ) -> error(req3b, 1) . delta
  ;

init V(init_gs);
```