Verified Design of an Automated Parking Garage

Aad Mathijssen A. Johannes Pretorius

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
The Netherlands

11th International Workshop on Formal Methods
for Industrial Critical Systems, 2006

Introduction

What is an automated parking garage?

In an automated parking garage, cars are parked fully automatically:

Doesn't work?

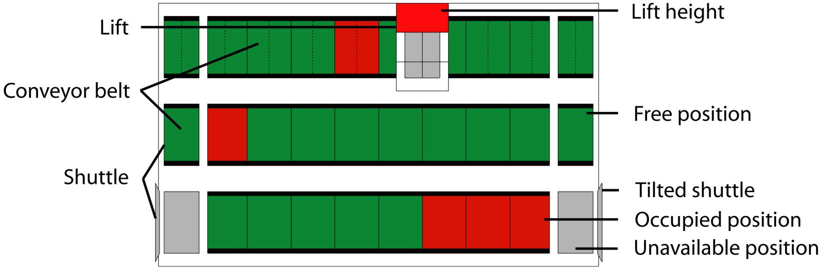
Outline

- Overview
  - The system at hand
  - Problem description
- Proposed solution
  - Conceptual design
  - Verification
- Points of interests
  - Visualization
  - Tech specs
- Conclusions

Overview

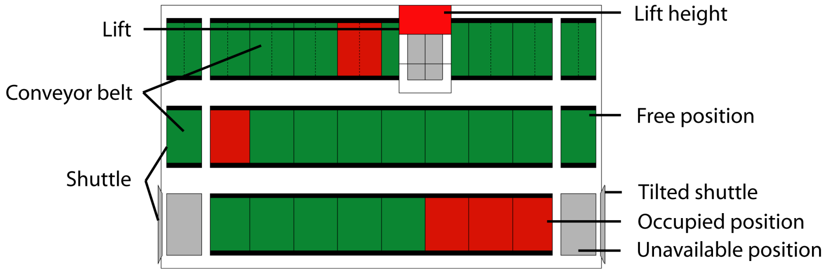
The system at hand (1)

Given is the following hardware configuration (floorplan):



Overview
The system at hand (1)

Given is the following hardware configuration (floorplan):



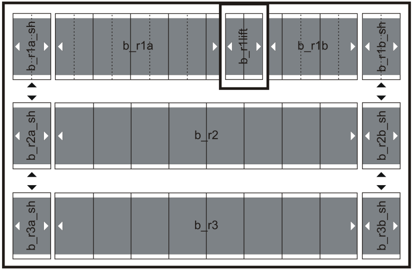
Complicating factors:

- awkward lift position: cars are able to move in half positions
- shuttles can be tilted

Overview

The system at hand (2)

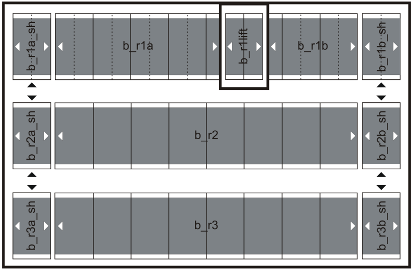
Given is the following hardware configuration (floorplan):



Overview

The system at hand (2)

Given is the following hardware configuration (floorplan):



Note that:

- arrows indicate movement of belts and shuttles
- there are 30 parking spots, maximum 29 occupied

Overview
Problem description

For this hardware configuration:

- design software

Overview

Problem description

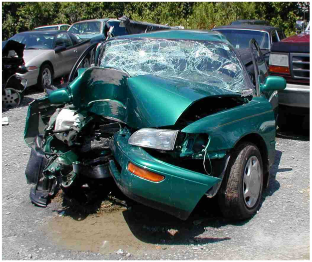
For this hardware configuration:

- design software
- such that safety is guaranteed

Overview
Problem description

For this hardware configuration:

- design software
- such that safety is guaranteed



Overview

Problem description

For this hardware configuration:

- design software
- such that safety is guaranteed

Proposed solution

Approach:

- create a high-level software design
- verify the design:
  - model this design
  - prove correctness of the model

Proposed solution

Approach:

- create a high-level software design
- verify the design:
  - model this design
  - prove correctness of the model

After creating the design, do not start implementation immediately. Instead, create a model:

- gain *insight* in the system
- detect *errors* in the proposed design
- foundation for *implementation*

Proposed solution

Approach:

- create a high-level software design
- verify the design:
  - model this design
  - prove correctness of the model

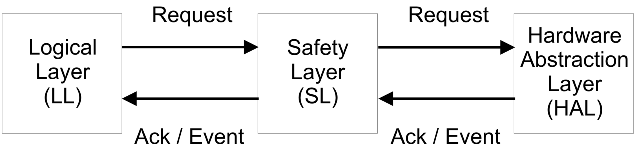
After creating the design, do not start implementation immediately. Instead, create a model:

- gain *insight* in the system
- detect *errors* in the proposed design
- foundation for *implementation*

Interactions are of primary concern: model behaviour.

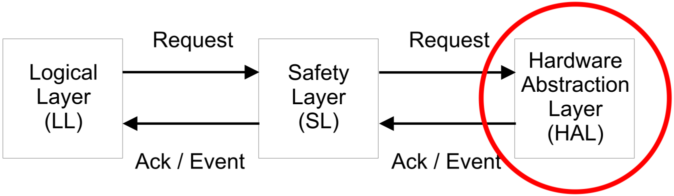
Conceptual Design
Architecture

Split the system in three layers:



Conceptual Design
Architecture

Split the system in three layers:

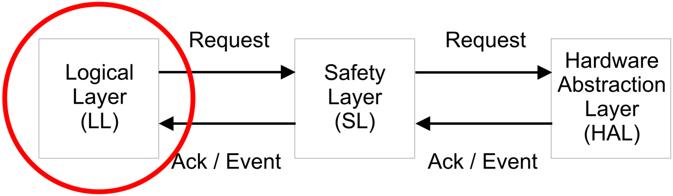


Hardware Abstraction Layer:

- abstract from hardware using instructions and events
- receive and execute instructions; provide feedback on results
- issue events to the other layers

Conceptual Design
Architecture

Split the system in three layers:

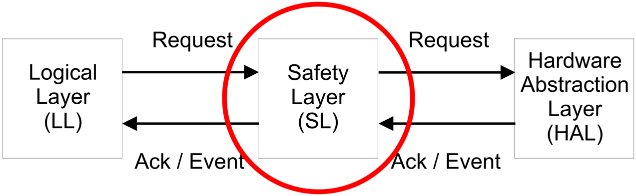


Logical Layer:

- the parking/retrieval algorithm
- issue the right instructions in the right order

Conceptual Design
Architecture

Split the system in three layers:



Safety Layer:

- pass messsages between the logical and hardware layer
- only if they are safe, deny otherwise

Conceptual Design
Data

The following data is communicated between the layers:

- *Instruction*: single instruction that the hardware should execute:
  - $move\_belts(bs, d, ms)$
  - $move\_shuttles(shs, o, d)$
  - $tilt\_shuttle(p, o)$
  - $move\_lift(h)$
  - $rotate\_lift$
- *InstructionSet*: instructions that are to be executed *concurrently* by the HAL
- *Result*: indicates the result of executing of a set of instructions
- *Event*: addition/removal of cars to/from the system

Conceptual Design

Interactions

The following interactions facilitate communication between the layers:

- $req(s)$: request of an instruction set s
- $ack\_req(s)$: acknowledgement of a request of an instruction set s
- $deny\_req(s)$: deny of a request of an instruction set s
- $ack\_exec(s, r)$: acknowledgement of execution of instruction set s with result r
- $occur(e)$: occurrence of an event e

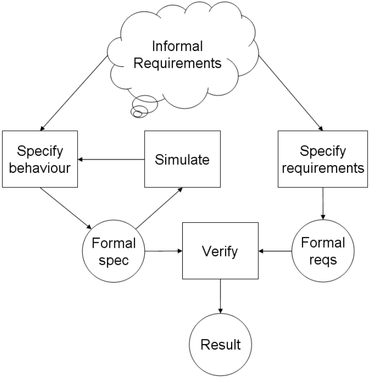
Verified Design
Focus and approach

We focus on the safety layer.

Verified Design
Focus and approach

We focus on the safety layer.

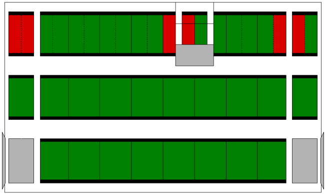
Our approach:



Informal Requirements of the Safety Layer

Examples of *safety* requirements the safety layer should meet:

1. If a car is moved between belts, both belts should move in the same direction.
2. Cars should not be able to move into walls.
3. When moving shuttles, cars may not be damaged.
4. When moving the lift, cars may not be damaged.



Behavioural specification of the Safety Layer

Behaviour

Behaviour of the safety layer:

- message passing of events and instruction sets
- acknowledge when:
  - a *set of instructions* is allowed
  - based on the *current state*

  deny otherwise

Behavioural specification of the Safety Layer

Allowed instruction sets

A set of instructions s is *allowed* if:

1. s specifies *at least one* instruction
2. the instructions in s *do not overlap*: the areas on which the instructions operate are pairwise disjoint
3. each individual instruction in s is allowed

Behavioural specification of the Safety Layer

Allowed instructions

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 halfway between the two outer belts of *bs* and their neighbours, if any.

Verification of the Safety Layer
Pros and cons

In general, verification:

- guarantees requirements are fulfilled for each possible system state
- requirements need to be formalised
- model checking is space and time consuming

Verification of the Safety Layer
Pros and cons

In general, verification:

- guarantees requirements are fulfilled for each possible system state
- requirements need to be formalised
- model checking is space and time consuming

Problem with the current specification: too many states and transitions

Verification of the Safety Layer

Pros and cons

In general, verification:

- guarantees requirements are fulfilled for each possible system state
- requirements need to be formalised
- model checking is space and time consuming

Problem with the current specification: too many states and transitions

Solution: apply reductions

Verification of the Safety Layer

Reductions (1)

Reductions we needed to apply:

- abstract from sets of instructions by focusing on *single* instructions on only
- abstract from requests and acknowledgements; instead, it is assumed that instructions are *executed successfully* by the HAL

Verification of the Safety Layer

Reductions (1)

Reductions we needed to apply:

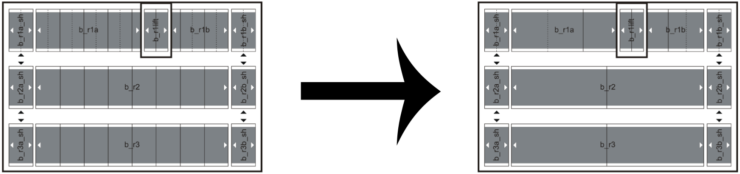
- abstract from sets of instructions by focusing on *single* instructions on only
- abstract from requests and acknowledgements; instead, it is assumed that instructions are *executed successfully* by the HAL

Results:

- simulation becomes possible
- verification still infeasible:
  state space of the model consists of 640 billion states
  $(6,4 * 10^{11})$

Verification of the Safety Layer
Reductions (2)

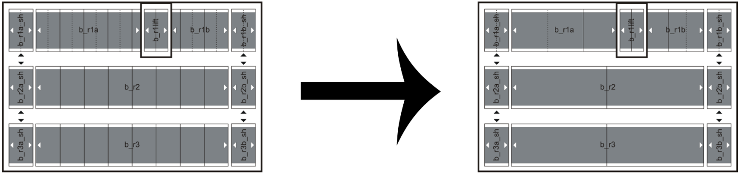
Restrict the number of *positions*:



Verification of the Safety Layer

Reductions (2)

Restrict the number of *positions*:



Result:

- 3,3 million ($3,3 * 10^6$) states and 98 million ($9,8 * 10^7$) transitions
- verification becomes feasible

Verification of the Safety Layer
Formalisation and application

Apply verification:

- formalise requirements:
  - express informal requirements as *enabling conditions* for illegal interactions
  - when an illegal interaction is possible, an *error* action is triggered
  - *augment* specification with error actions
- check the state space on the existence of the *error* actions

Verification of the Safety Layer

Formalisation and application

Apply verification:

- formalise requirements:
  - express informal requirements as *enabling conditions* for illegal interactions
  - when an illegal interaction is possible, an *error* action is triggered
  - *augment* specification with error actions
- check the state space on the existence of the *error* actions

Result: no *error* actions were found

Visualization

Detecting errors

Verification found no errors in the proposed design.

Errors were found:

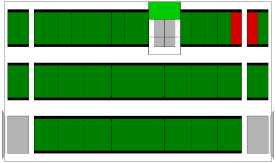
- mostly during specification and simulation
- the tricky ones using a custom built visualization plugin to the simulator

Visualization
Detecting errors

Verification found no errors in the proposed design.
Errors were found:

- mostly during specification and simulation
- the tricky ones using a custom built visualization plugin to the simulator

Problem with the shuttles:

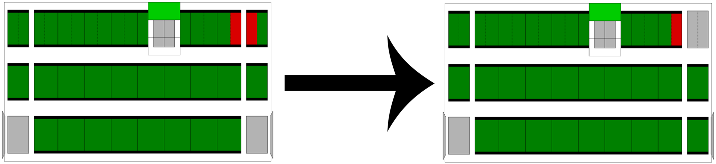


Visualization
Detecting errors

Verification found no errors in the proposed design.
Errors were found:

- mostly during specification and simulation
- the tricky ones using a custom built visualization plugin to the simulator

Problem with the shuttles:



Visualization

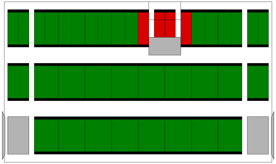
Detecting errors

Verification found no errors in the proposed design.

Errors were found:

- mostly during specification and simulation
- the tricky ones using a custom built visualization plugin to the simulator

Problem with the lift:

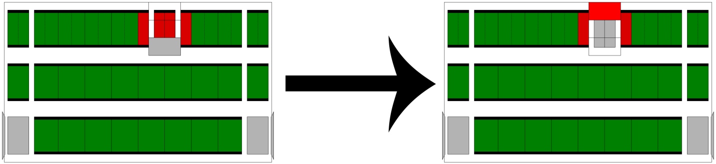


Visualization
Detecting errors

Verification found no errors in the proposed design.
Errors were found:

- mostly during specification and simulation
- the tricky ones using a custom built visualization plugin to the simulator

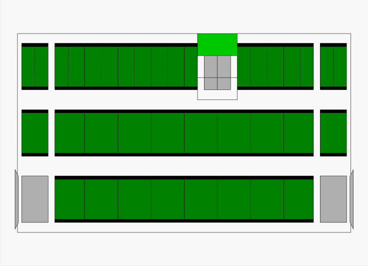
Problem with the lift:



Visualization
Advantages

Advantages of the visualization plugin:

- revealed *errors* in the model
- *speeds up* simulation
- enhances *communication*



Doesn't work?

Tech Specs

The verification tool we used is mCRL2:

- combines process algebra with higher-order abstract data types
- the successor of μCRL

Tech Specs

The verification tool we used is mCRL2:

- combines process algebra with higher-order abstract data types
- the successor of μCRL

Lines of code:

- specification: 991 lines of mCRL2 code
- verification: 217 lines of mCRL2 code
- visualization: 1583 lines of C++ code

Tech Specs

The verification tool we used is mCRL2:

- combines process algebra with higher-order abstract data types
- the successor of μCRL

Lines of code:

- specification: 991 lines of mCRL2 code
- verification: 217 lines of mCRL2 code
- visualization: 1583 lines of C++ code

Verification time (real time):

- 5 hours on a cluster of 34 CPUs (3 GHz CPU, 2 GB RAM)
- 35 hours on a single PC (3 GHz CPU, 4 GB RAM)

Tech Specs

The verification tool we used is mCRL2:

- combines process algebra with higher-order abstract data types
- the successor of μCRL

Lines of code:

- specification: 991 lines of mCRL2 code
- verification: 217 lines of mCRL2 code
- visualization: 1583 lines of C++ code

Verification time (real time):

- 5 hours on a cluster of 34 CPUs (3 GHz CPU, 2 GB RAM)
- 35 hours on a single PC (3 GHz CPU, 4 GB RAM)

Time spent: approximately 500 man hours

Conclusions
Using formal methods

In general, the use of formal methods *bridges the gap* between functional requirements and an actual implementation.

Model the system:

- gain *insight* in the system
- detect *errors* in the design
- foundation for *implementation*

Simulation: *confidence* in our model

Verification: *prove* correctness of our model

Conclusions

The automated parking garage case study

Positive results in this case study:

- *layered* system design where each layer has its own task
- the safety layer has been *proven* correct
- the model can be *implemented* in software almost directly
- effective use of *vizualisation* techniques

Conclusions

The automated parking garage case study

Positive results in this case study:

- *layered* system design where each layer has its own task
- the safety layer has been *proven* correct
- the model can be *implemented* in software almost directly
- effective use of *vizualisation* techniques

Negative findings:

- the current hardware setup is *not optimal*
- there lies a *performance challenge* in the real system

Further reading

- Mathijssen, A., Pretorius, A.J.:
  Specification, analysis, and verification of an automated parking garage.
  Technical Report 05-25, Technische Universiteit Eindhoven (2005)
- www.mcrl2.org:
  mCRL2 homepage.

What if the Movies do not work?

Screenshots (1)

In an automated parking garage, cars are parked fully automatically:




Back

What if the Movies do not work?

Screenshots (2)

The plugin speeds up simulation and enhances communication:

Back