Modelchecking Non-Functional Requirements for Interface Specifications

Florian Kammüller and Sören Preibusch

Institut für Softwaretechnik und Theoretische Informatik

FIT'08 Budapest, 5. April 2008

Software Ubiquitous in Safety Critical Systems

Software Ubiquitous in Safety Critical Systems

Importance of Specification and Verification of rich Interfaces

Our Approach: Requirements Feedback Loop

- Feedback loop between requirements and design phases
  1. Requirements: Object-Z specification M (functional), Temporal Logic formulas ρ (non-functional)
  2. Translate to SMV: $\phi(M), \phi(\rho)$
  3. Modelcheck result (Consistency and $\phi(M) \vdash \phi(\rho)$)
  4. Use feedback for improvement of requirements and iterate

Overview

1. Background
2. Translation from Object-Z to SMV
3. Case Study TWIN-Elevator
4. Feedback Loop: Example
5. Conclusions

Specification in Object-Z

- Extension of specification language Z by objects:
  - OO-features: Classes, objects
  - Interfaces (by visibility lists)
  - Inheritance, polymorphism
- Formal specification of Interfaces in a very general way
- Full predicate calculus available to specify invariants and operations

$\Rightarrow$ Rich interfaces (for functional requirements) can be explicitly specified

$\Rightarrow$ Non-functional requirements integrated through temporal logic and feedback loop

SMV Modelchecking

ACM Turing Award Honors Founders of Automatic Verification Technology

Researchers Created Model Checking Technique for Hardware and Software Designers

- SMV is a classical Modelchecker for CTL and LTL
- SMV's input language similar to programming language (Verilog style)
- Supports simple datatypes and modules

Translation from Object-Z to SMV

- *Profit from syntactic similarities*
  - ⇒ Translation identifies: types(bool), constants, classes, instantiation
- *Watch out for semantical differences*
  - ⇒ Special: state transition, operations, operation composition

plus	minus
Δ(var)	Δ(var)
var' = var + 1	var' = var - 1

```
next(var) := var + 1;          next(var) := var - 1;
```

⇒ SMV: `"redefining var"`

Special Translation for Operations: Stimulus

In O-Z operations are offered, must be invoked by environment
Additional variable mimicks operation stimulus explicitly in SMV

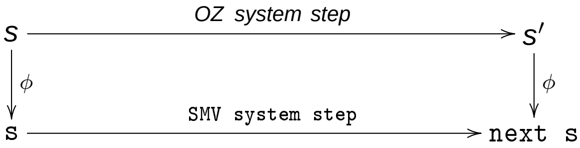
```
minus
Δ(var)
────────
var > 0
var' = var - 1
```

```
minus_pre : boolean;
minus_pre := (var > 0);
```

```
next(var) := case {
  plus_stimulus & plus_pre : var + 1;
  minus_stimulus & minus_pre : var - 1;
  default: var; };
```

Translation from Object-Z to SMV

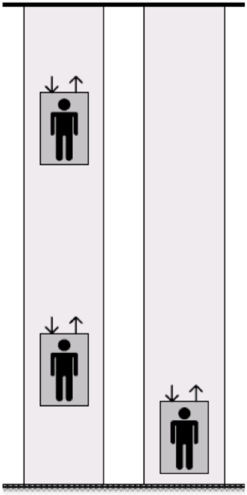
- Translation Soundness
  - Translation is a homomorphism ϕ of the boolean lattices of Object-Z and SMV
  - State change preserves ϕ: both formalism contain logical representation of "change"
    - In Object-Z: pre-state s, post-state s'
    - In SMV: pre-state `s`, post-state `next s`

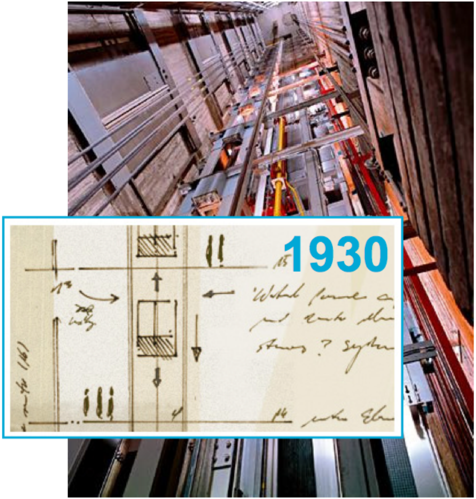


- Translation *not* complete:
  - Quantification only for finite domains
  - OO notion of self not supported by SMV modules

. . . but sufficiently practical

The TWIN Problem and ThyssenKrupp Solution





Feedback Loop: Example from TWIN

Cabin

$\upharpoonright (curr\_level)$

$curr\_level : Level$

INIT

$curr\_level = LevelGround$

MoveUp

$\Delta(curr\_level)$

$curr\_level' = curr\_level + 1$

- Example requirement: cabin's movements are restricted by boundaries shaft:

```
LevelGround ≤system.curr_level ≤ LevelTop
```

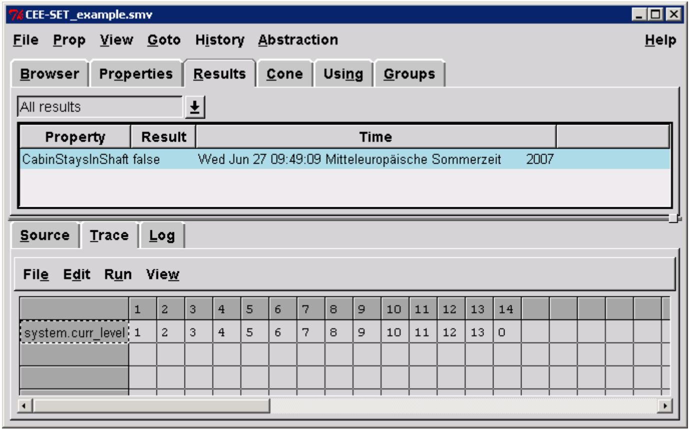
- Translation into SMV:

```
CabinStaysInShaft: assert
  G (LevelGround <= system.curr_level)
 & (system.curr_level <= LevelTop);
```

[cont'd]

Feedback Loop: Example from TWIN [cont'd]

- SMV detects violation of safety requirement:



⇒ Operation `MoveUp` responsible for illegal change of `curr_level`

⇒ Add precondition `curr_level < LevelTop`

Results of the Case Study

- All safety properties of TWIN system could be successfully verified
- Observations:
  - Feedback loop enables detection of inconsistencies
  - Specification is abstract description of interfaces reducing to behaviour relevant for safety, e.g. DSC

actual implementation of cabin selection is irrelevant

⇒ this part of the behaviour is *not* part of interface

Summary and Lessons Learned

- Summary:
  - Interface specification with Object-Z
  - Feedback loop to requirements for quality:
    - Translation to SMV
    - Modelcheck specification
  - Case study TWIN-elevator shows feasibility
- Object-Z: formal specification “traditional way”
  - Stepwise, iterated specification refinement
  - Elegant and abstract (in comparison to SMV)
  - Temporal logic together with Object-Z seems natural
  - Pragmatism of translation; so far no restriction