# A Model-Based Development Process for Embedded Systems

Ninth Bieleschweig Workshop, Hamburg, May 2007

Maritta Heisel

Joint work with Denis Hatebur

http://swe.uni-duisburg-essen.de
Email: {Maritta.Heisel,Denis.Hatebur}@uni-due.de

Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft

# DePES I

- Concrete process for developing embedded systems.
- Consisting of 12 steps, including
  - requirements analysis
  - system architecture
  - software architecture
  - component specification and implementation
  - systematic testing
- Each step results in some document(s).
- Expressed mostly in UML notations.
- Validation conditions for checking coherence between documents.

# DePES II

- Developed over time and gradually improved in an industrial context.
- Based on development processes for security- and safety-critical systems according to the Common Criteria and IEC 61508.
- Emerged from projects, e.g., development of smartcard operating systems and applets for smartcards, and motor control and automatic doors.
- For a complete description, see [Hat06].

# Principles underlying DePES

- Clear terminology
- Thorough environment modeling
- Stress on problem analysis
- Pattern usage
  - problem frames
  - architectural styles
  - code patterns
- Model-based development
  - develop sequence of models, each describing different aspects of the system/machine
  - models can be analyzed and checked for coherence
- Explicit process description with validation conditions
- Systematic testing

# Terminology [Jac01]

Machine thing we are going to build; may consist of software and hardware

Environment part of the world where the machine will be integrated

System consists of machine and its environment

Requirements optative statements; describe how the environment should behave when the machine is in action

Specification implementable requirements; describe the machine; are basis for its construction

Domain knowledge indicative statements; consist of facts and assumptions; needed to derive specification

# Phases of DePES I

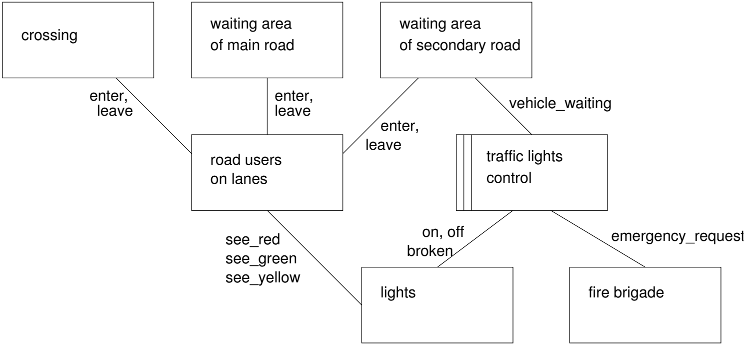
- Problem definition
  - Jackson-approach [Jac01] using problem frames
  - dependencies between subproblems are made explicit
  - specifications expressed using UML 2.0 sequence diagrams
- System design
  - system architecture defined using UML 2.0 composite structure diagrams
- Software design
  - layered architecture
  - extended four-variable model
  - merge software architectures from subproblems
- Component specification
  - sequence diagrams for component interface
  - class diagrams and UML 2.0 state machines for component description

# Phases of DePES II

- Software implementation
  - coding patterns based on state machines available for Java
- Integration and testing
  - testing against sequence diagrams set up in earlier phases
  - new: use state-machine approach to test against requirements

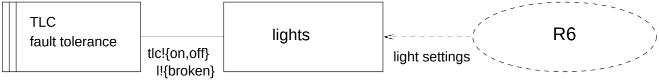
# State problem, describe environment

- State requirements
- State facts and assumptions
- Model the environment using a context diagram



# Decompose problem: problem diagrams

- Decompose problem into simple subproblems



R6 In case of a broken light bulb the traffic lights should blink in yellow for the secondary road, after all red lights have been switched on for a period of time.

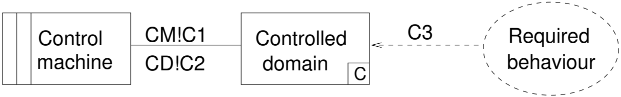
- Specify dependencies between subproblems: sequential, alternative, parallel

$$
\begin{array}{lll}
< start > & ::= & (< main\_passing > \,||\, < fire > \,||\, < broken\_light >) \\
< main\_passing > & ::= & (MainRoadPassing < sec\_passing >) \\
< sec\_passing > & ::= & (SecondaryRoadPassing < main\_passing >) \\
< fire > & ::= & EmergencyRequestSecondaryRoadPassing \\
 & & < main\_passing > \\
< broken\_light > & ::= & BrokenLightSafeState
\end{array}
$$

# Fit subproblems to problem frames

Problem frames

- Are patterns for simple development problems
- Fitting a problem to some problem frame means instantiating the frame diagram
- Example: required behaviour problem frame

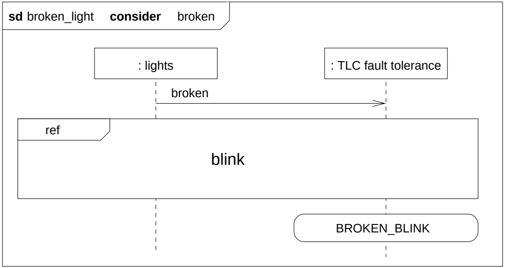


- Problem of previous slide is instance of required behaviour

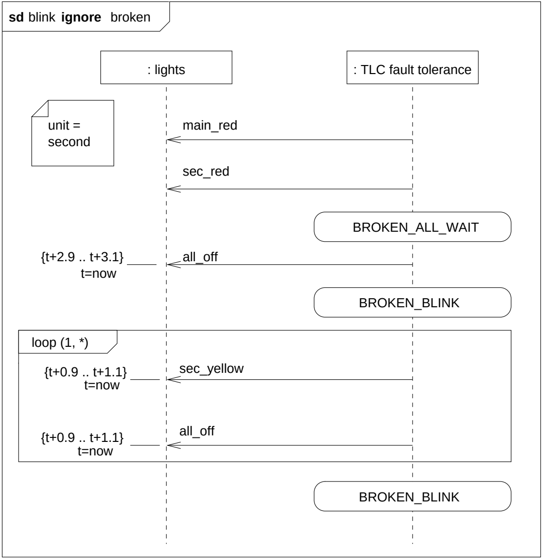
# Transform requirements into specifications I

For each subproblem:

- Use domain knowledge to transform non-implementable requirements into specifications [JZ95].
- Express the specifications as sequence diagrams.
- Validation condition: signals in sequence diagrams must be the same as phenomena in machine interface of problem diagram.

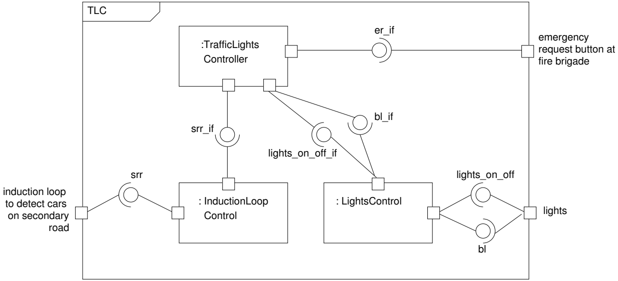


# Transform requirements into specifications II



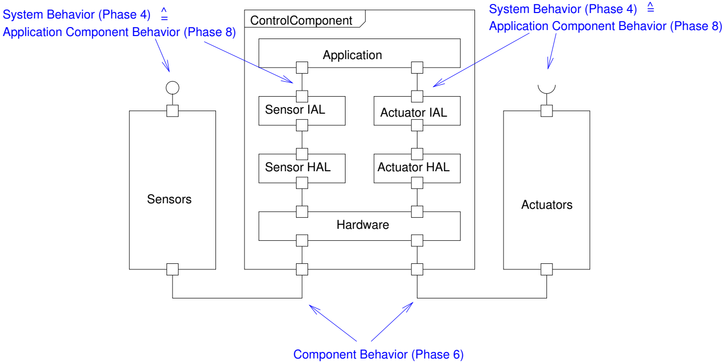
# Set up system architecture

- System architecture consists of hardware and software components
- Notation: UML composite structure diagrams
- Interface behavior of all programmable components must be specified using sequence diagrams
- Validation condition: to each programmable component, at least one subproblem must be associated.


TLC
:TrafficLights Controller
er_if
emergency request button at fire brigade
srr_if
bl_if
lights_on_off_if
srr
induction loop to detect cars on secondary road
: InductionLoop Control
: LightsControl
lights_on_off
lights
bl

# Software design: layered software architecture

- Basic idea: application layer software should have the the same interfaces as the machine, i.e., monitored and controlled variables [BH99].
- Thus, application layer becomes device-independent, device dependencies are factored out in IALs and HALs.



System Behavior (Phase 4) ≙ Application Component Behavior (Phase 8)

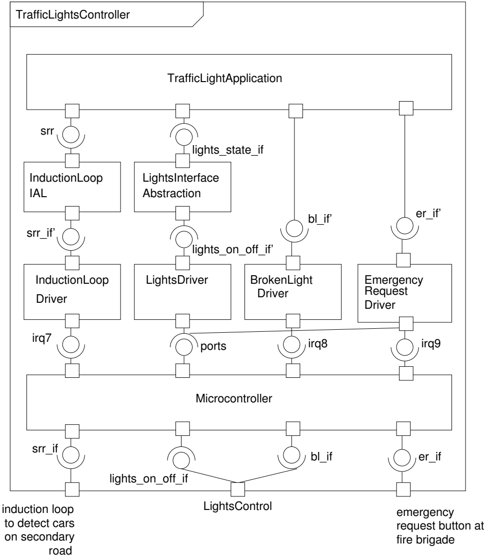
System Behavior (Phase 4) ≙ Application Component Behavior (Phase 8)

Component Behavior (Phase 6)

# Architectural patterns, global software architecture

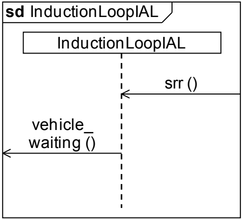
- We have defined a layered architecture for each of Jackson's problem frames [CHH05].
- Hence, for each subproblem fitted to a problem frame, we get a (preliminary) software architecture.
- The global software architecture is defined by merging the subproblem architectures according to rules based on the subproblem dependencies (problem definition phase) [CHH06].

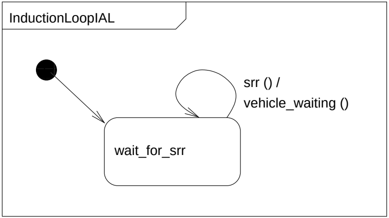
# Traffic light control: global software architecture



# Component specification

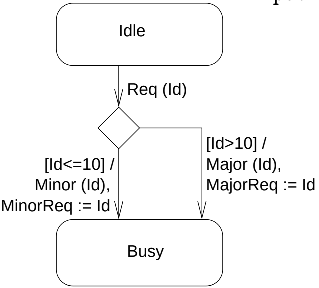
- Interface specification using sequence diagrams
- Component description using interface classes and state machines
- Validation conditions: state machine must be complete and generate the behavior stated in the sequence diagrams





# Component implementation: pattern for Java

```
public class ComponentName implements provided_if {
    static final int IDLE = 0, BUSY = 1;
    private req_if ri;
    private int state;
    ...
    public ComponentName (req_if ri_) {
        state = ... // Init state
        ri = ri_;
    }
    public void Req(int id) {
        switch (state) {
            case IDLE:
                if (id<=10) {
                   if (ri!=NULL) ri.Minor (id); ....
                   state = BUSY;
                } else {
                   if (ri!=NULL) ri.Major (id); ....
                   state = BUSY;
                }
                break;
            default:
                assert false: "FSM error Req";
        }
    } ...
```



# Quality assurance

Achieved by

- Checking validation conditions as specified in process descriptions
- Systematic testing

Systematic testing:

- Develop test cases during earlier phases of the development, i.e., *before* the implementation
- Test against *requirements* also, not only against specification
- For this purpose: model environment by stochastic processes (work in progress)

# DePES: all steps I

1. Describe problem
2. Consolidate requirements
3. Decompose problem
4. Derive a machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture

# DePES: all steps II

7. Design an architecture for all programmable components of the global system architecture that will be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. Integrate and test hardware and software

# Relating software phases of DePES to Common Criteria and ISO/IEC 61508 I

| **CC documents** | **61508 process step** | **DePES** |
|---|---|---|
| ST (Security Target) | Software/E/E/PES safety requirements specification | 1-4 |
| ADV_FSP (Functional Specification) | Specification part of Software/E/E/PES safety requirements specification | 6 |
| ADV_ARC (Security Architecture) | Software architecture | 7 |
| ADV_TDS (TOE Design) | Software system design / Module design | 8-9 |
| ADV_IMP (Implementation representation) | CODING | 10 |

ADV = **A**ssurance class for **D**e**v**elopment

# Relating software phases of DePES to Common Criteria and ISO/IEC 61508 II

| **CC** | **61508** | **DePES** |
|---|---|---|
| ATE_DPT (Depth) | Module and Integration testing | 10 |
| ATE_FUN (Functional Tests) | Integration and validation testing | 11 |

ATE = **A**ssurance class for **Te**sting
ATE_COV (Coverage) and ATE_IND (Independent testing) are not explicitly given in 61508 and DePES, but are part of the respective testing phases.
DePES phases 5 and 12 are not mapped since these phases consider hardware.

# What do we gain by defining such a process? I

**Fact**

DePES is not a light-weight process!

- Certification according to safety- and security standards (IEC 61508 and Common Criteria) is supported.
- Sequence of well-defined steps helps developers to focus attention on relevant parts of the task (and fake a rational design process ;-).
- Developed models and their interrelations can be checked in each step.
- Special attention is paid to the analysis phase and the modeling of the environment. (Environment models yield test cases.)

# What do we gain by defining such a process? II

- Non-functional (quality) characteristics can be taken into account (in particular, safety and security; by specific architectures and problem frames).
- Problem decomposition is performed explicitly and systematically. Relations between subproblems are exploited to compose partial solutions of subproblems.
- Using patterns in various phases support re-use of existing knowledge and (partial) automation.
- Various possibilities for tools support:
  - UML tools available.
  - Tool for generating sequence diagrams available.
  - Model checker for UML state machines available.
  - Other tools conceivable.
- Process emerged from industrial practice, uses well-established languages and techniques.

# Literature I

Ramesh Bharadwaj and Constance Heitmeyer.
Hardware/Software Co-Design and Co-Validation using the SCR Method.
In *Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99)*, 1999.

Christine Choppy, Denis Hatebur, and Maritta Heisel.
Architectural patterns for problem frames.
*IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures*, 152(4):198–208, 2005.

Christine Choppy, Denis Hatebur, and Maritta Heisel.
Component composition through architectural patterns for problem frames.
In *Proc. XIII Asia Pacific Software Engineering Conference*, pages 27–34. IEEE Computer Society, 2006.

# Literature II

- Denis Hatebur.
  A pattern- and component-based process for embedded systems development.
  Master's thesis, University Duisburg–Essen, 3 2006.
  http://swe.uni-duisburg-essen.de/intern/dpes.pdf.
- Michael Jackson.
  *Problem Frames. Analyzing and structuring software development problems*.
  Addison-Wesley, 2001.
- Michael Jackson and Pamela Zave.
  Deriving specifications from requirements: an example.
  In *Proc. 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.