Model-driven Development of Self-organizing Control Applications (MODOC)

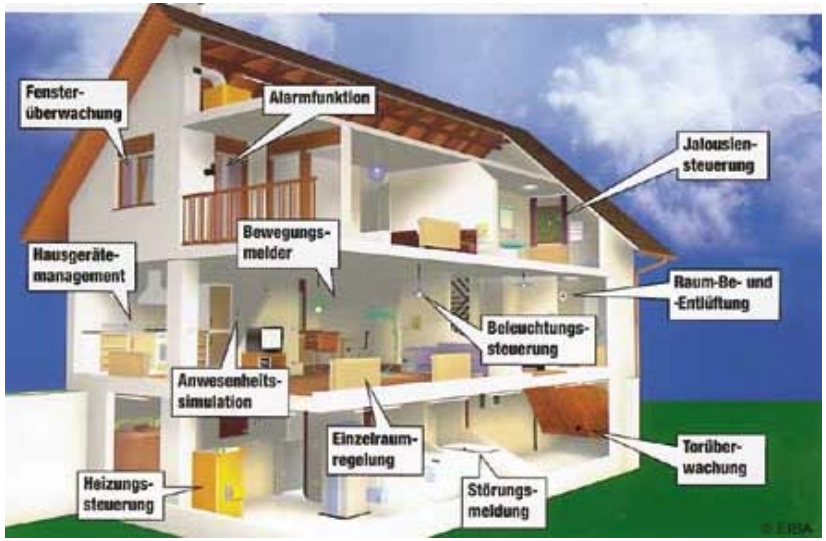
Prof. Dr.-Ing. Torben Weis
Dr. Arno Wacker
Dipl.-Inform. Sebastian Holzapfel
Dipl.-Inform. Christopher Boelmann
Universität Duisburg-Essen

Prof. Dr. Hans-Ulrich Heiß
Dr.-Ing. Jan Richling
Dipl.-Ing. Arnd Schröter
Technische Universität Berlin

Prof. Dr.-Ing. Gero Mühl
Dipl.-Inform. Helge Parzyjegla
M.Sc. Enrico Seib
Universität Rostock

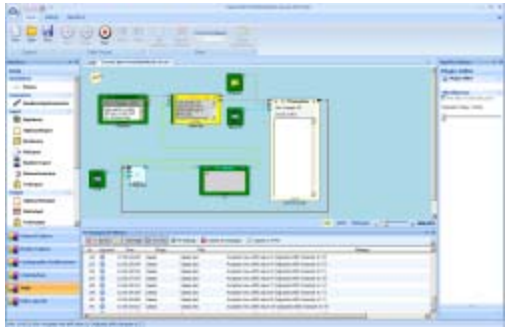
Self-* in Embedded Systems

- „Pervasive Computing at Large“
  - Tiny computers in day-to-day devices
    - Clothing,
    - Kitchen devices,
    - Buildings, …
- Self-Organization
  - Manual administration is impossible
  - Tiny devices are mobile and not very reliable



- Self-Stabilization
  - External sources can induce transient errors in the hardware
    - Radio noise, solar radiation, voltage fluctuation, …
  - Cost pressure on hardware manufacturing makes tiny computing devices less reliable, too

Software Development Methodology for OC



Application model

Back-annotation

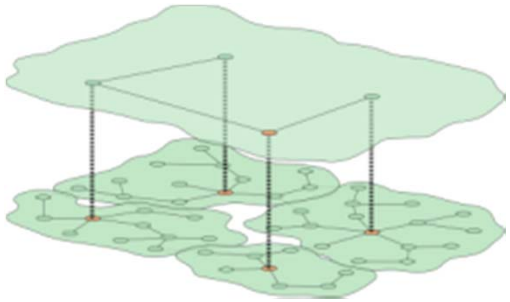
> Monitoring interfaces

> Reflection

Self-X compiler

> Self-stabilizing algorithms

> Self-stabilizing Turing Machine



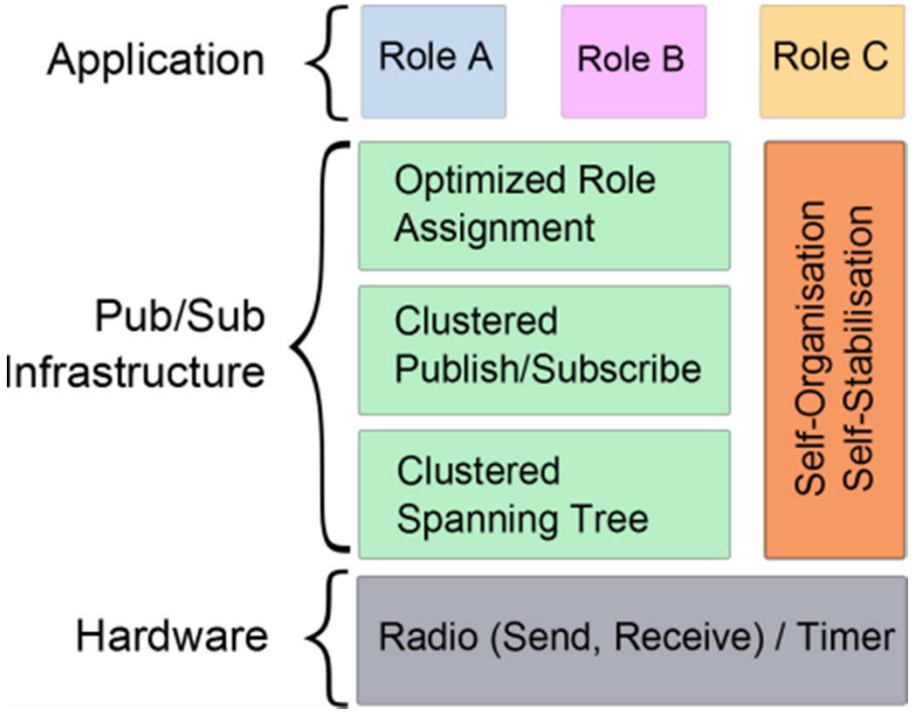
Stack | Heap | Code | Watchdog

Self-stabilizing Machine

Self-stabilizing and safe implementations

Overview

- Self-Stabilizing Controller
  - Embedded systems
  - MSP 430 controller
- Self-X Composition
  - Composition challenge
  - Example and evaluation
- Composite Events
  - Distributed detection
  - Self-organizing detector placement
- Conclusions



Towards a Self-stabilizing Controller

- Self-stabilizing automaton
  - Adaptation of a 3-tape Turing Machine with I/O capabilities
  - Addition of an energy concept to force the decay of old data
  - This has shown theoretical feasibility
- Self-stabilizing virtual machine
  - Stack machine approach, similar to Java or .NET
  - Self-stabilizing data structures
  - Assurance: After a transient fault, code is executed correctly again after a bounded time
  - Approach: A watchdog resets the machine if the main loop is not reached in time
- Self-stabilization on the MSP 430 controller
  - Realization of above assurance on real hardware

Application Anatomy

- Anatomy of a networked sensor/actuator application

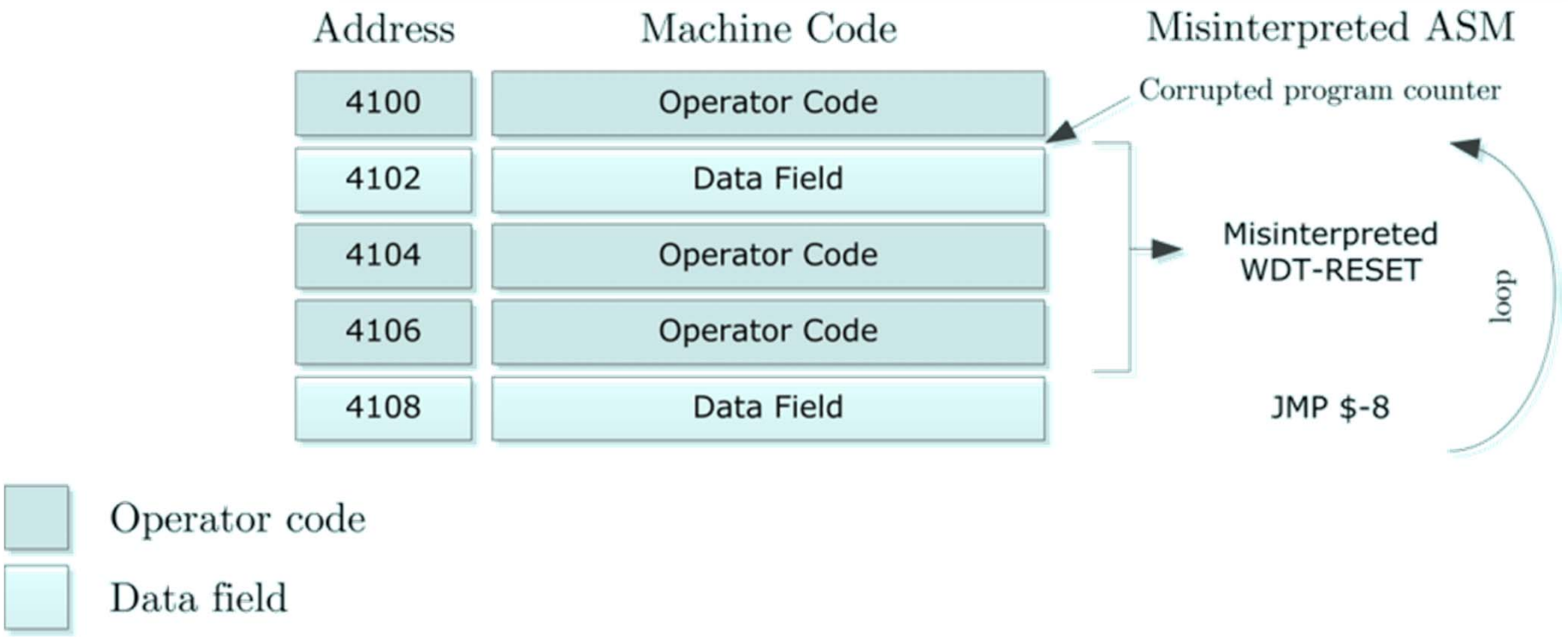
```
void main() {
   while(true) {
      ev = wait_for_event();
      process_event(ev);
      send_output();
      reset_watchdog();
} }
```

- For self-stabilization, the software has to return to the main loop in bounded time in any case after a transient fault
- Some faults can invalidate this assumption …

MSP 430 Controller

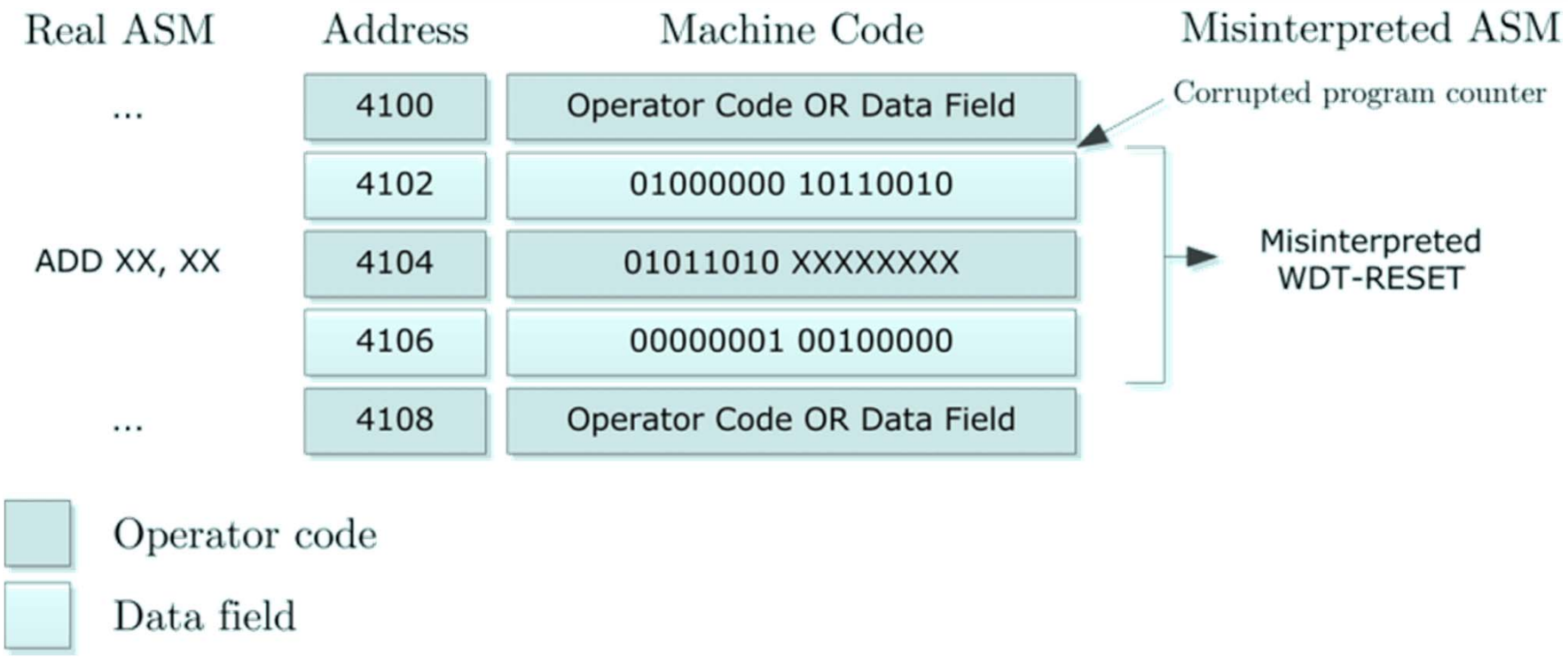
- Frequently used for embedded systems and sensor networks
- Possible temporary faults
  - Bit errors in RAM (tackled by our self-stabilizing virtual machine)
  - **Bit errors in CPU registers**
  - **Faulty execution of CPU instructions**
- Execution of unintended CPU instructions
  - PC register points to the data field of a CPU instruction
  - This may lead to an unintended infinite loop
  - However, the watchdog timer can rescue the system
- Worst case scenario
  - Unintended infinite loop resets the watchdog in each iteration
  - Self-stabilization would no longer be possible

Unintended Infinite Loops



- Unintended loop is entered due to corrupted PC register
- This loop is only critical if it continuously resets the watchdog

Unintended Watchdog Reset

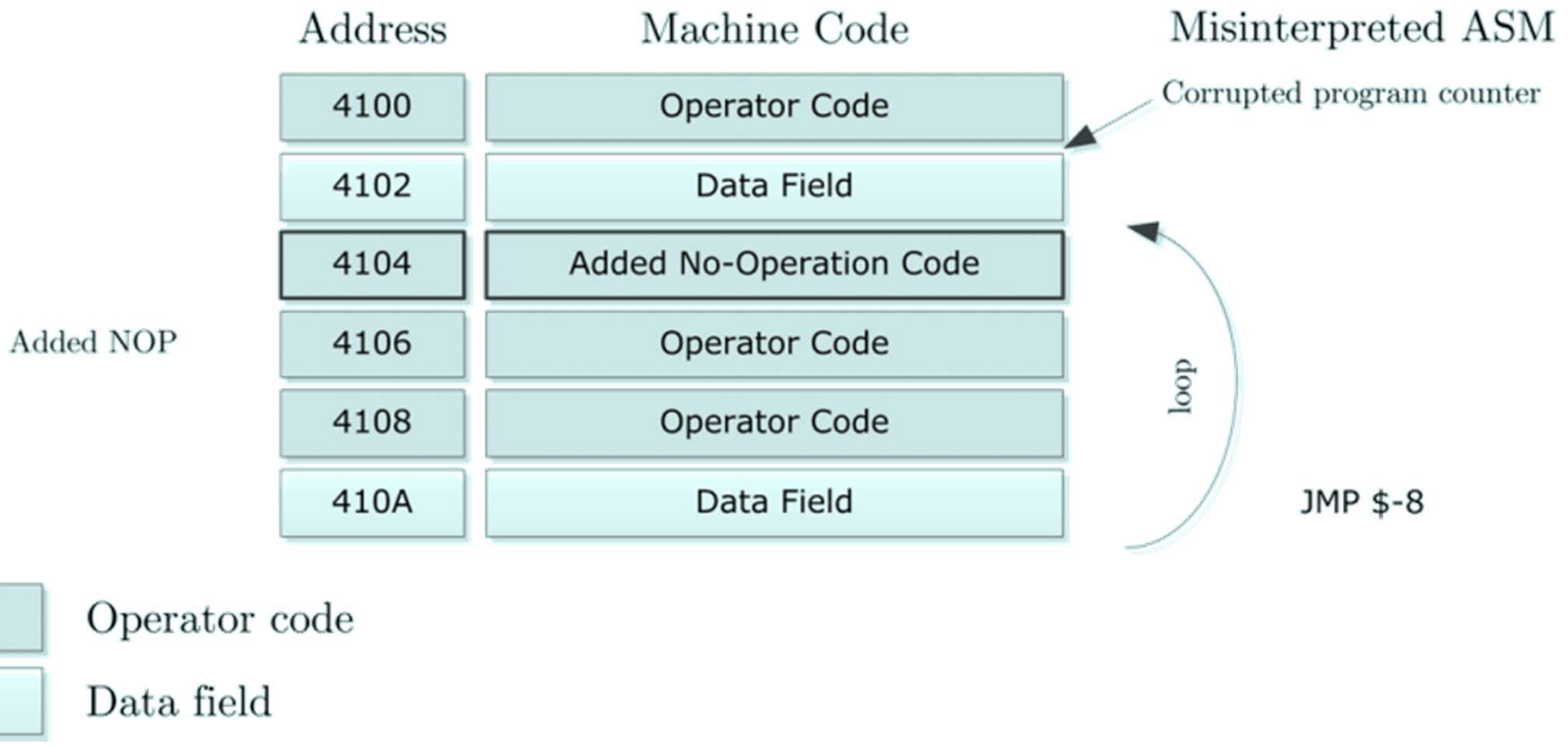


- Correct execution: Opcodes at 0x4100, 0x4104, 0x410a
- Unintended execution: Opcode at 0x4102 (watchdog reset) followed by an unintended JMP at 0x4108

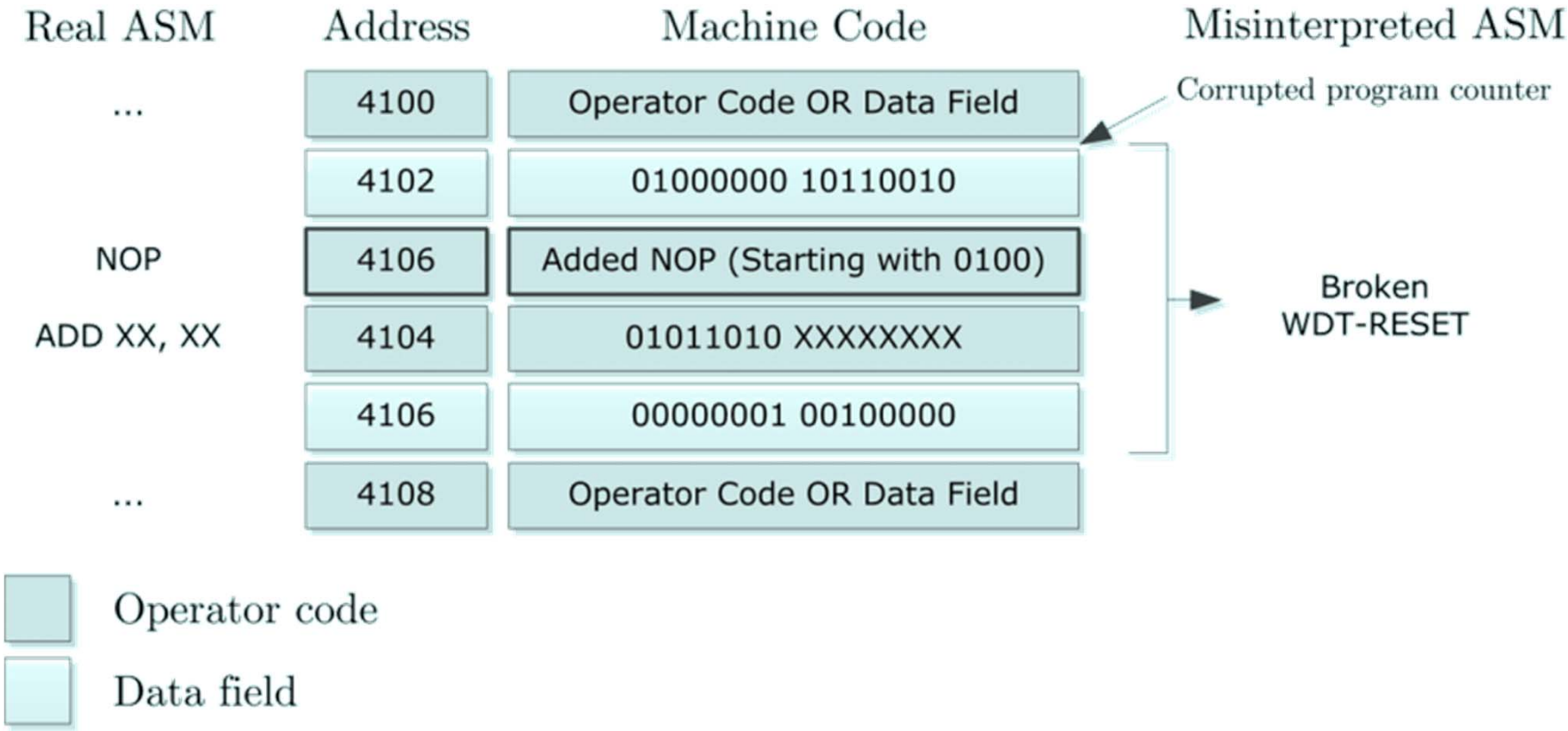
Solutions

- Unintended loop detection
  - Find all possible unintended loops
  - Ensure that the CPU returns to correct instructions eventually by inserting NOPs
  - An inserted NOP ensures that an unintended JMP targets the NOP instead of the data field of an operation
- Optimized unintended loop detection
  - Cure only loops which reset the watchdog
  - Limits the number of inserted NOPs
- Watchdog protection
  - Prevent unintended code from resetting the watchdog
  - Elegant solution, but not possible for all hardware architectures
  - Possible for the MSP 430 (again by inserting NOPs)

Breaking Unintended Infinite Loops



Prevention of Unintended Watchdog Reset



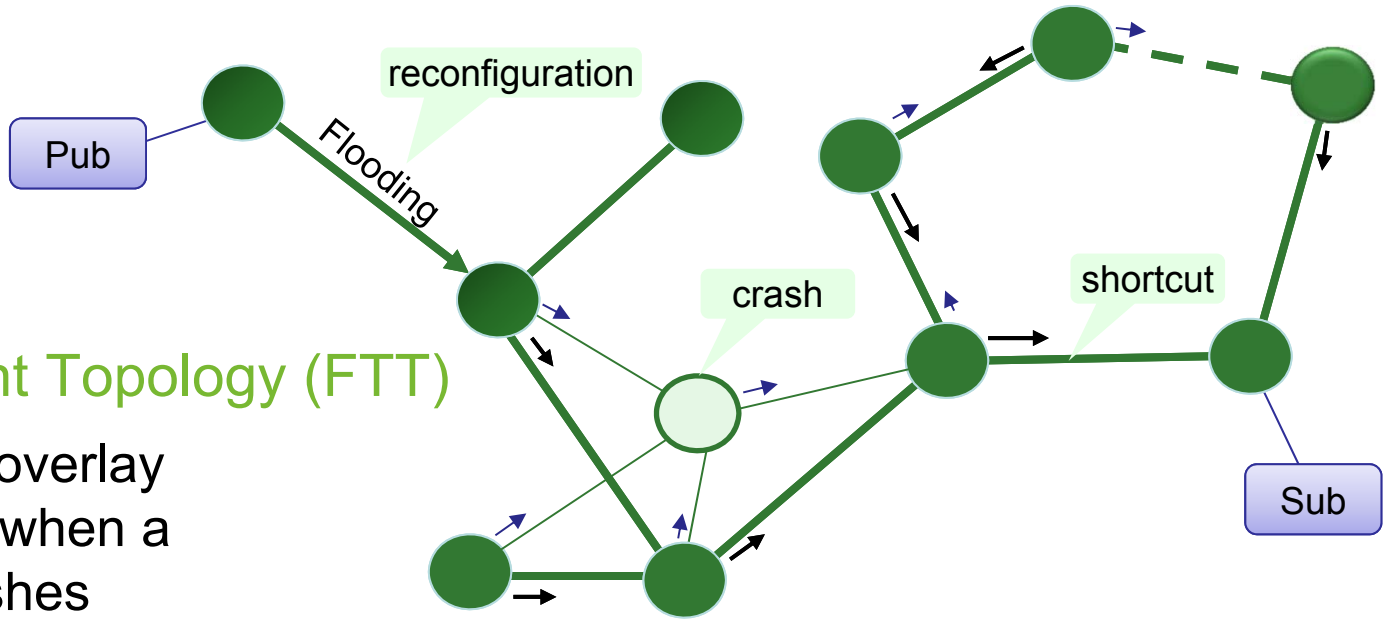
Self-X Algorithms

Self-Optimizing Routing (SOR)

> Switches between different routing schemes for each link (flooding vs. filtering)

Self-Optimizing Topology (SOT)

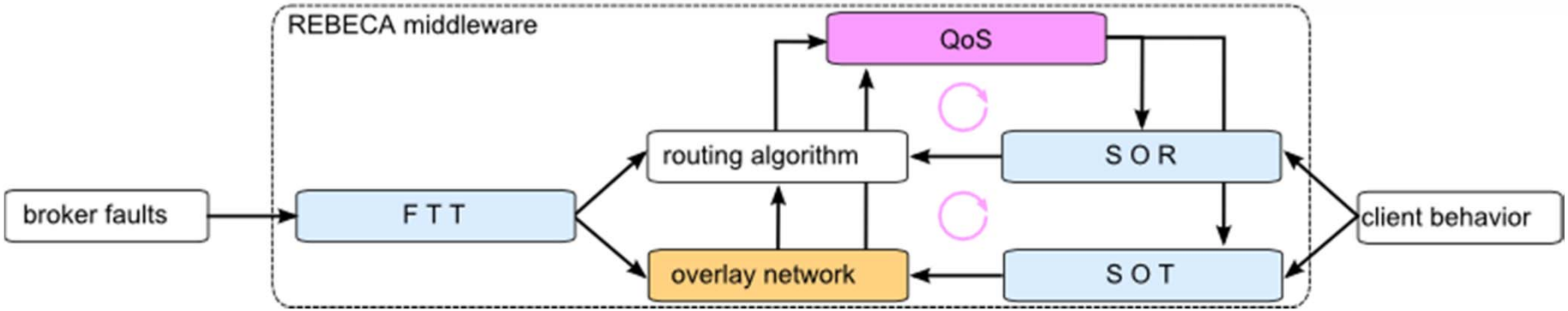
> Connects sub-topologies with similar interests



Fault Tolerant Topology (FTT)

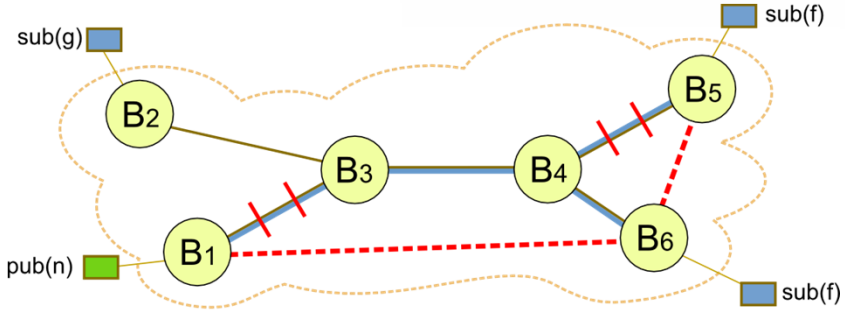
> Keeps the overlay connected when a broker crashes

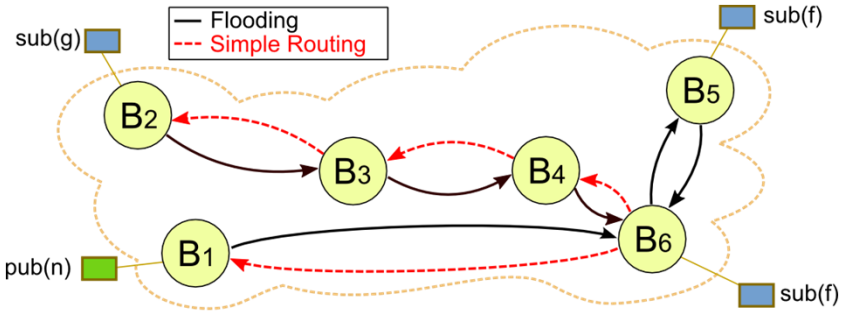
Composition of Self-X Algorithms

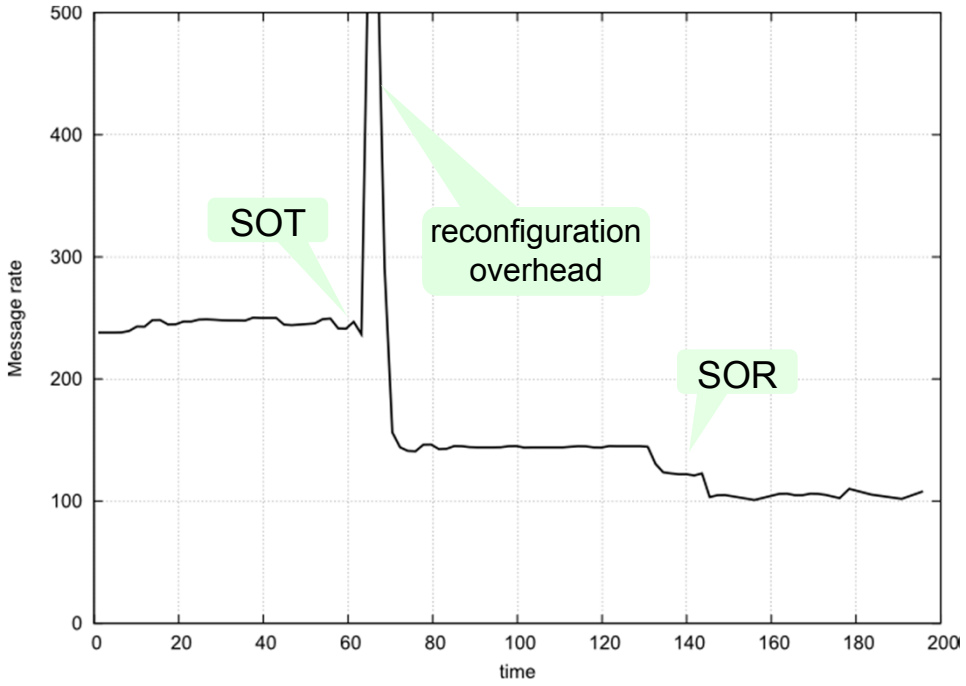


- Superimposed QoS feedback loops
- Dependency analysis shows conflict in overlay reconfiguration
- Conflict Resolution
  - Transaction scheme for SOT to deal with broker crashes (FTT)
  - Support for connecting arbitrary topologies with SOR
  - Mutual blocking of SOT and SOR

Composition of Self-X Algorithms







- Preserves properties of composed algorithms
- Achieves higher performance than each single algorithm

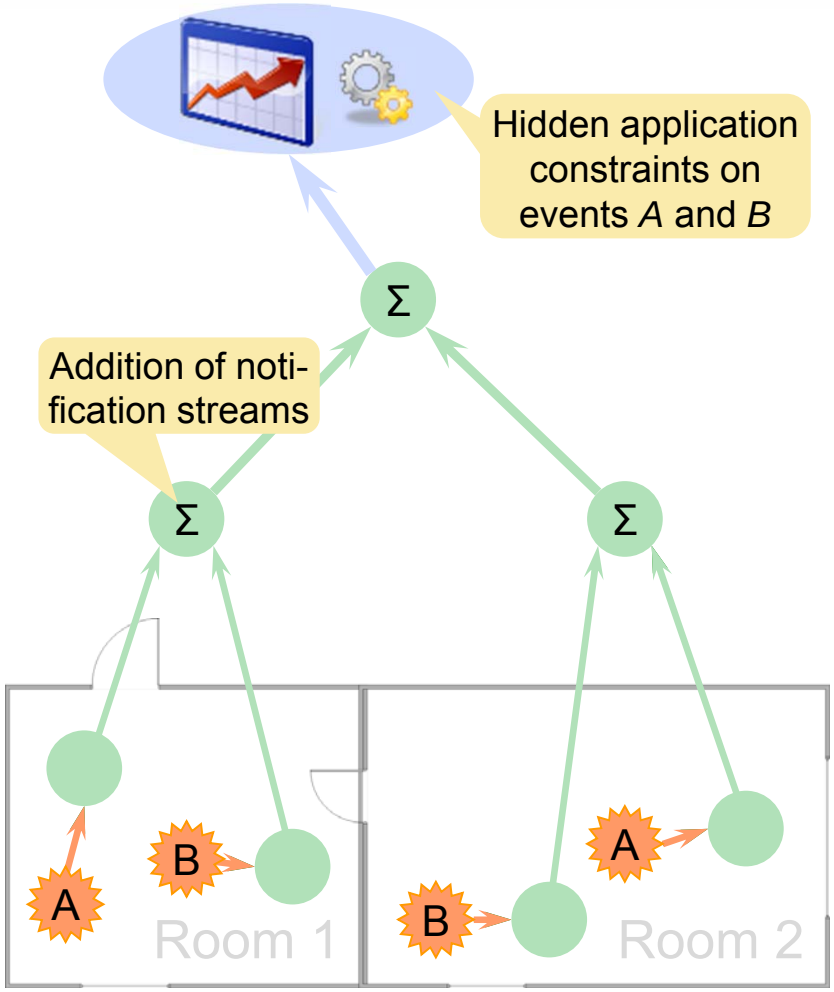
Event Patterns

Event patterns

- Application roles communicate by exchanging notifications via publish/subscribe
- Actions are often triggered only if several conditions are met → event patterns

Application level detection of event patterns

- Candidate notifications must be forwarded to application
- Notification traffic concentrates at event sinks → bottlenecks
- Majority of notifications are forwarded unnecessarily



Composite Event Detection

- Event composition at middleware level
  - Replaces/complements pattern recognition at application level to enable efficient distributed pattern detection
  - Patterns are specified by composition algebra (definition, visibility, reusability)
- Four basic detector operations based on composition algebra used for optimization

Decomposition

- Hierarchical decomposition of a detector into constituent subpatterns

Recombination

- Dissolve no longer useful detectors and recombine them with others

Migration

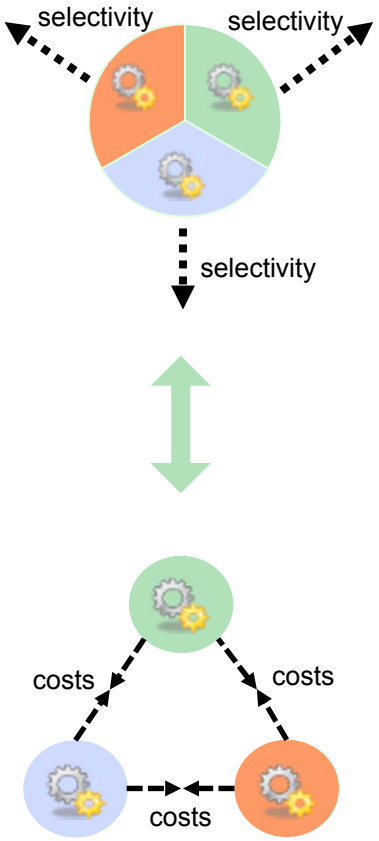
- Early filtering by seamless migration of detectors along the event stream

Replication

- Divide event space into disjoint domains

Force Model

- Heuristic based on relaxing forces due to dynamically changing environment
  - Gradually optimizes placement using local knowledge and basic detector operations
  - Balances responsiveness and stability
- Model system as compensating forces



Selectivity

- Indicates potentially saved forwarding costs
- Pull detectors towards sources or sinks
- Migration or decomposition with replication depends on the number of pulling forces

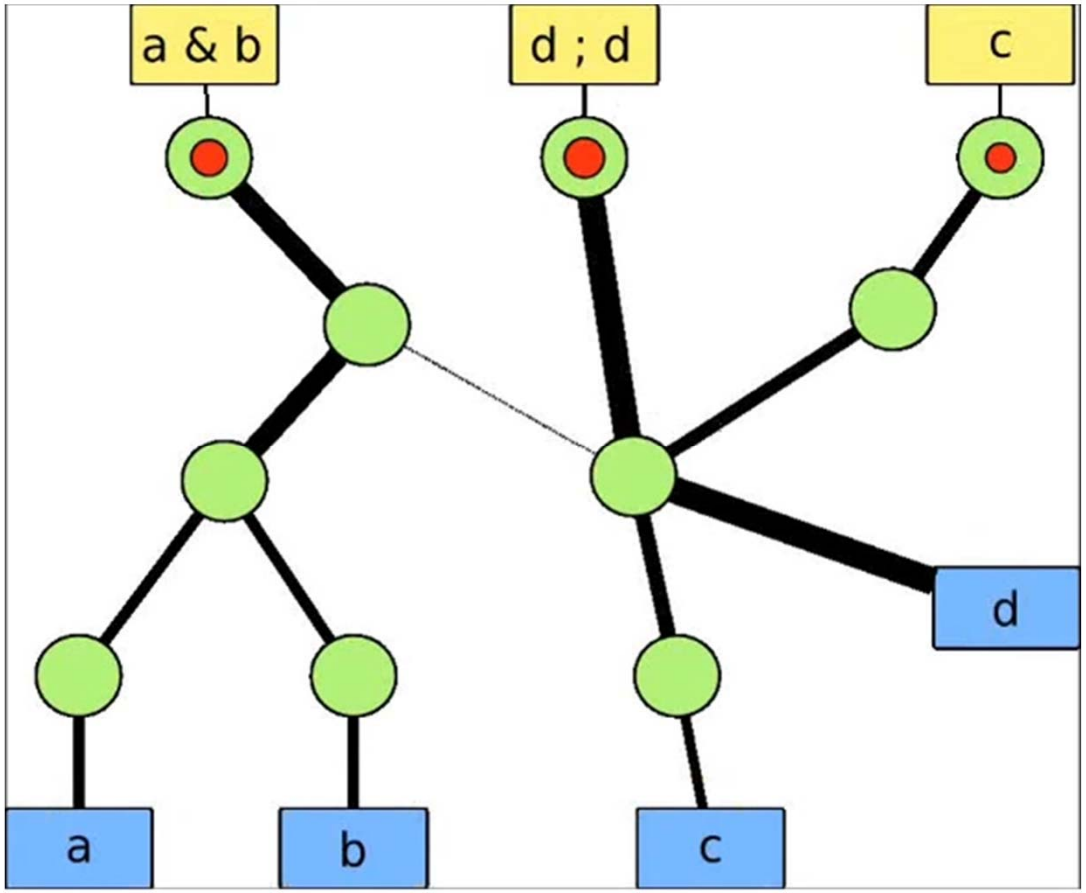
Costs

- Storage utilization, migration costs
- Pull related detectors together → recombination

Friction

- Counter oscillations but keep system responsive

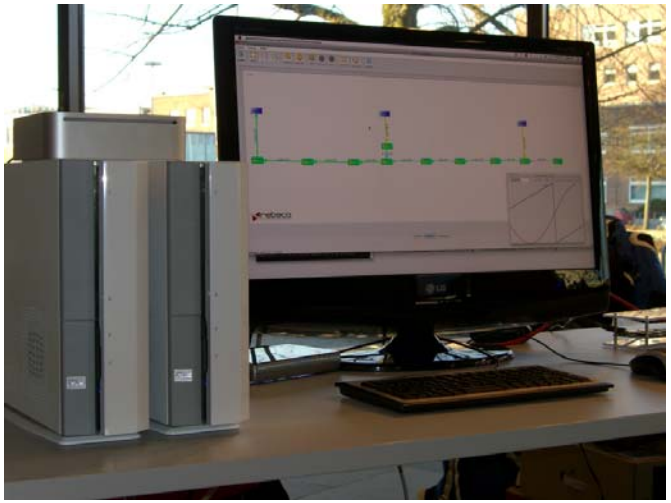
Simulation



- Discrete event simulation of detector placement strategy
- Network consumption → width of lines
- Computational load → area of red dots
- Shows decomposition, migration and recombination

Conclusions

- MODOC allows for engineering of distributed control-applications
  - Comprehensive tool chain supporting modeling, code generation, deployment and debugging of OC applications
- MODOC provides self-organization and self-stabilization
  - Self-stabilizing controller and virtual machine
  - Self-organizing and self-optimizing publish/subscribe infrastructure



Discussion

Thanks for your kind attention.

Prof. Dr. Gero Mühl

Architecture of Application Systems

University of Rostock

gero.muehl@uni-rostock.de

http://wwwava.informatik.uni-rostock.de