# Model Driven Software Development in Service Robotics

Prof. Dr. Christian Schlegel

*Computer Science Department*
*University of Applied Sciences Ulm*

http://www.zafh-servicerobotik.de/ULM/index.php
http://www.hs-ulm.de/schlegel

AG Schlegel:
M.Sc. Siegfried Hochdorfer, B.Sc. Alex Lotz, B.Sc. Matthias Lutz, B.Sc. Dennis Stampfer, B.Sc. Andreas Steck

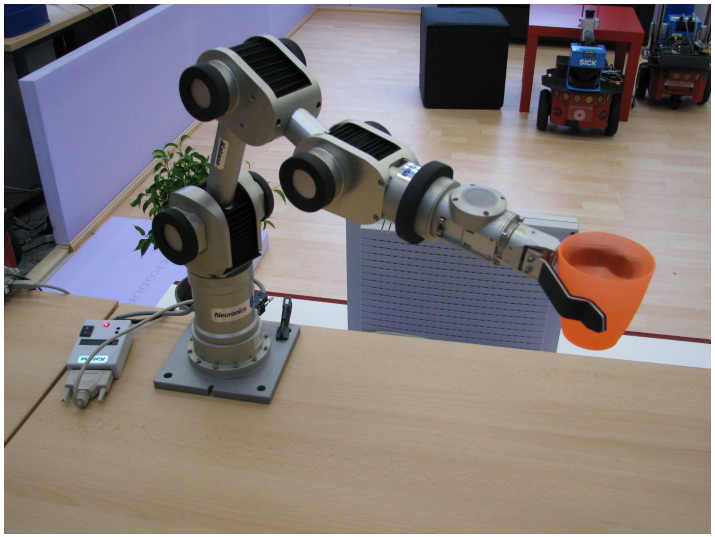
Hochschule Ulm

service robotik

# AG Schlegel / Ulm

## What are we doing ?

- embedded realtime and autonomous systems
  - software technology / MDSD for service robotics
- probabilistic approaches
  - low-cost bearing-only SLAM / SURF
  - visual odometry

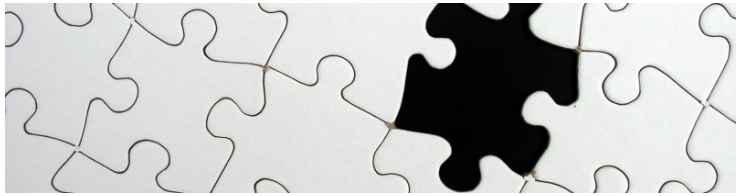




Hochschule Ulm

# Model Driven Software Development
# Introduction and Motivation

## What is this talk about ?



- not just another software framework
- not just another middleware wrapper

➔ we have plenty of those ...

## But

- separation of robotics knowledge from short-cycled implementational technologies
- providing sophisticated and optimized software structures to robotics developers not requiring them to become a software expert

## How to achieve this ?

- make the step from code-driven to model-driven designs
- there are open source tools useful in robotics !

# Model Driven Software Development
# Introduction and Motivation

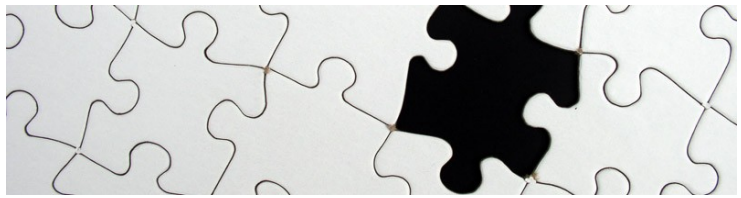
## Why is Model Driven Software Development important in Robotics ?

- get rid of hand-crafted single unit service robot systems
- compose them out of standard components with explicitly stated properties
- be able to reuse / modify solutions expressed at a model level
- take advantage from the knowledge of software engineers that is encoded in the code transformators
- be able to verify properties (or at least provide conformance checks)

- be able to address resource awareness !!

and many many more good reasons

**Engineering the software development process in robotics is one of the basic necessities towards industrial-strength service robotic systems**



Hochschule Ulm

# Model Driven Software Development
# Introduction and Motivation

## What is different in robotics ?

- ***not*** the huge number of different sensors and actors
- ***not*** the various hardware platforms
- ***not*** real-time requirements etc.

***but***

- the context and situation dependant reconfiguration of interactions
- a prioritized assignment of restricted resources to activities again depending on context and situation

## vision for the next steps in robotics:

- resource-awareness at all levels to be able to solve tasks adequately given certain resources

# Model Driven Software Development
# Idea and Approach

## That sounds good but give me an example ...

we made some very simple but pivotal decisions:

- granularity level for system composition:
  - loosely coupled components
  - services provided and required
- strictly enforced interaction patterns between components
  - precisely defined semantics of intercomponent interaction
  - these are policies (and can be mapped onto any middleware mechanism)
  - separate component internals from externally visible behavior

  ➔ *independent of a certain middleware*

  ➔ *enforce standardized service contracts between components*
- minimum component model to support system integration
  - dynamic wiring of the data flow between components
  - state automaton to allow for orchestration / configuration

  ➔ *ensures composability / system integration*

# Model Driven Software Development
# Idea and Approach

## That sounds good but give me an example ...

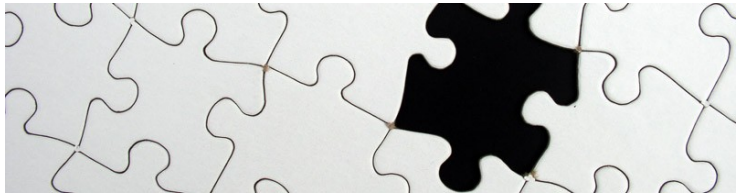
- execution environment
  - tasks (periodic, non-periodic, hard real-time, no realtime), synchronization, resource access

  ➔ *components explicitly allocate resources like processing power and communication bandwidth from the underlying HAL*

  ➔ *again, can be mapped onto different operating systems*
- design policy for component behavior:
  - principle of locality: a component solely relies on its own resources
  - example: QUERY
    - maximum response time attribute of service provider
    - client can only *ask* (attribute in request object) for faster response as guaranteed by QoS of service provider
    - server would respond with a VOID answer in case of a reject of timeline
    - it is the client that then decides what is next
  - example: PUBLISH/SUBSCRIBE
    - service provider agrees upon QoS as soon as subscription got accepted
    - again, it is a matter of policy whether this already requires hard guarantees or whether we also accept notifications about not being able to hold the schedule

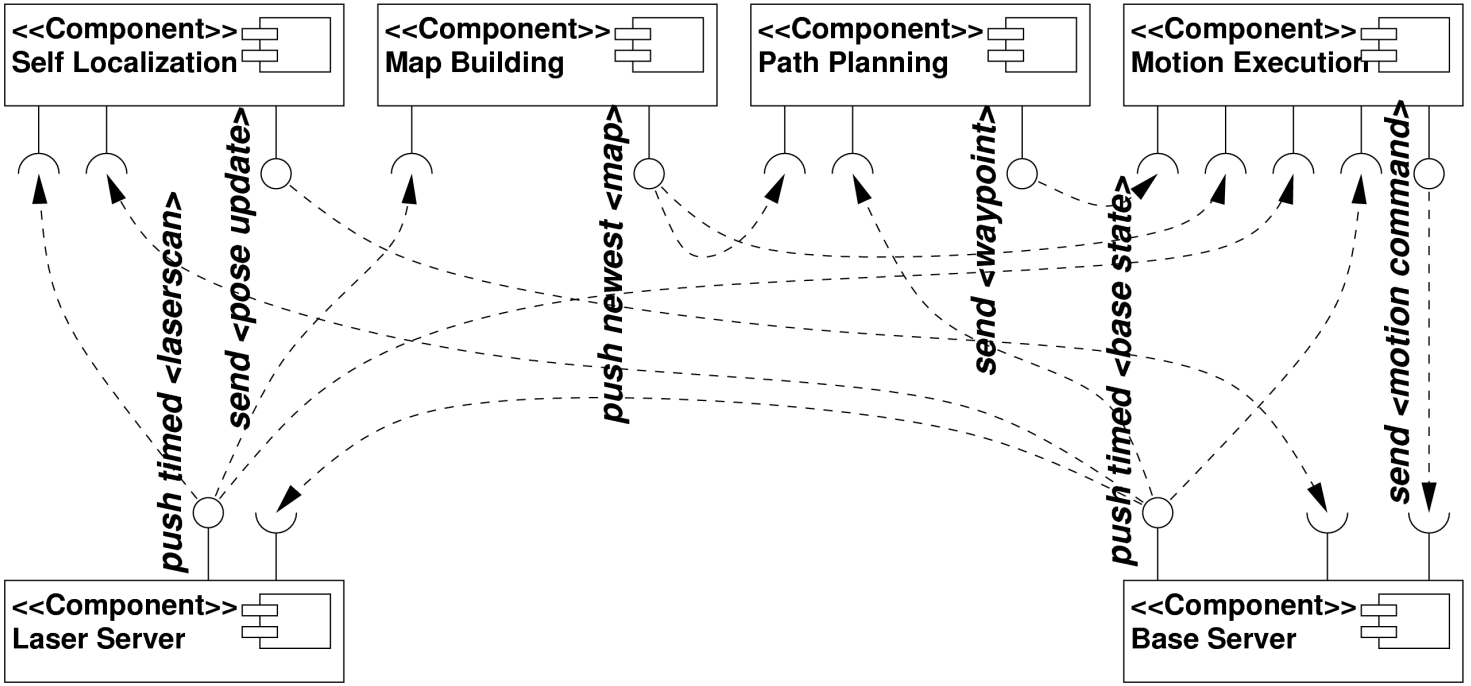
# Model Driven Software Development
# Focus Today / Workshop

- Introduce Toolchain based on Open Architecture Ware
  - fully integrated into Eclipse
  - http://www.openarchitectureware.org/
- Introduce MDSD Toolchain Example
  - PIM: SmartMARS robotics profile (Modeling and Analysis of Robotics Systems)
  - PSM: SmartSoft in different implementations but with the same semantics !
  - can be easily adapted to different profiles / profile extensions / PSMs
- Short Summary on SmartSoft [LGPL]
  - http://smart-robotics.sourceforge.net/
  - http://www.zafh-servicerobotik.de/ULM/en/smartsoft.php
  - CORBA (ACE/TAO) based SmartSoft
    - on web with various robotics components
  - ACE (without CORBA) based SmartSoft
    - under stress testing
    - available within July / August [Linux, Windows, ...] on sourceforge
  - oAW Toolchain for SmartSoft
    - available within July / August on sourceforge



service robotik

# Model Driven Software Development
## Idea and Approach




| | |
|---|---|
| send | CHS::SendClient, CHS::SendServer, CHS::SendServerHandler |
| query | CHS::QueryClient, CHS::QueryServer, CHS::QueryServerHandler |
| push newest | CHS::PushNewestClient, CHS::PushNewestServer |
| push timed | CHS::PushTimedClient, CHS::PushTimedServer, CHS::PushTimedHandler |
| event | CHS::EventClient, CHS::EventHandler, CHS::EventServer, CHS::EventTestHandler |
| wiring | CHS::WiringMaster, CHS::WiringSlave |

Hochschule Ulm

# Model Driven Software Development
## Idea and Approach

R
A

**Query Client**

+ QueryClient(:SmartComponent*) throw(SmartError)
+ QueryClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ QueryClient(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ *~QueryClient() throw() [virtual]*

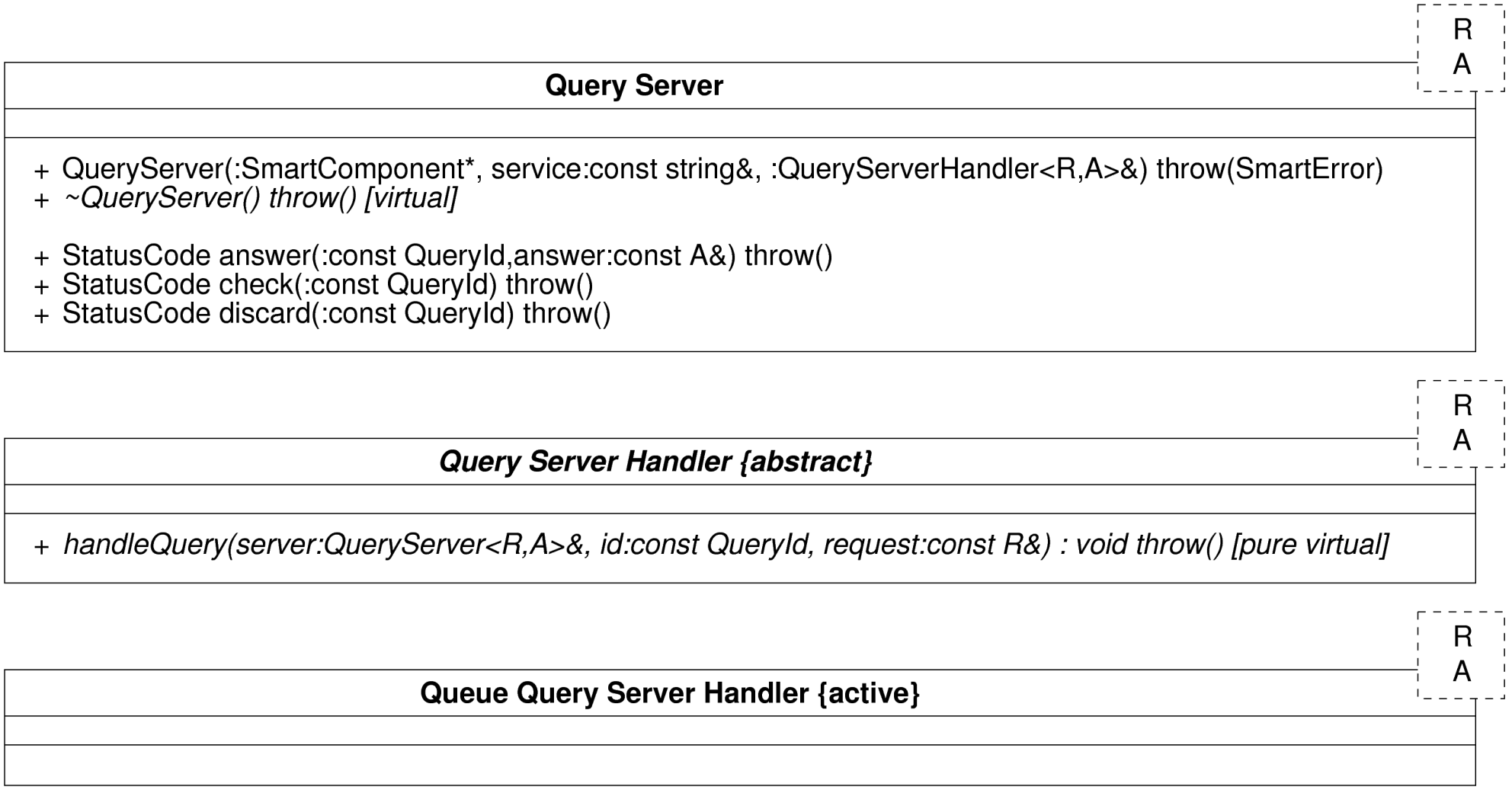
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()

+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()

+ blocking(flag:const bool) : StatusCode throw()

+ query(request:const R&, answer:A&) : StatusCode throw()
+ queryRequest(request:const R&, id:QueryId&) : StatusCode throw()
+ queryReceive(id:const QueryId, answer:A&) : StatusCode throw()
+ queryReceiveWait(id:const QueryId, answer:A&) : StatusCode throw()
+ queryDiscard(id:const QueryId) : StatusCode throw()

# Model Driven Software Development
## Idea and Approach

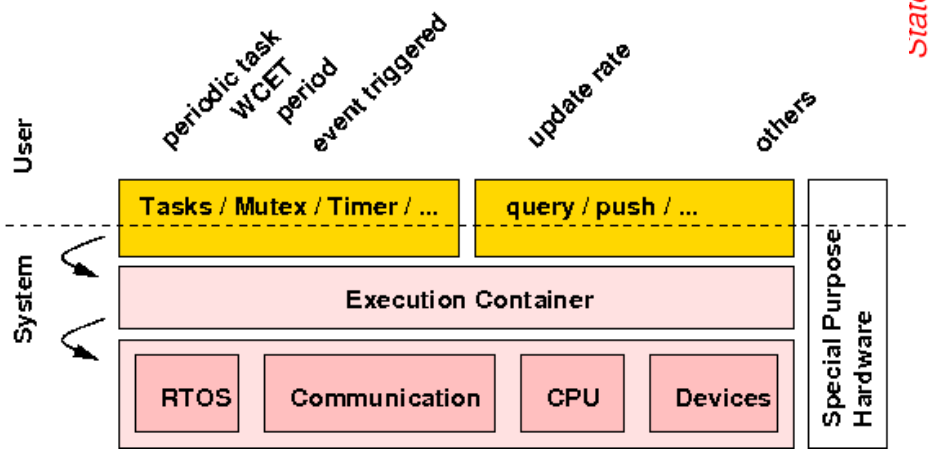


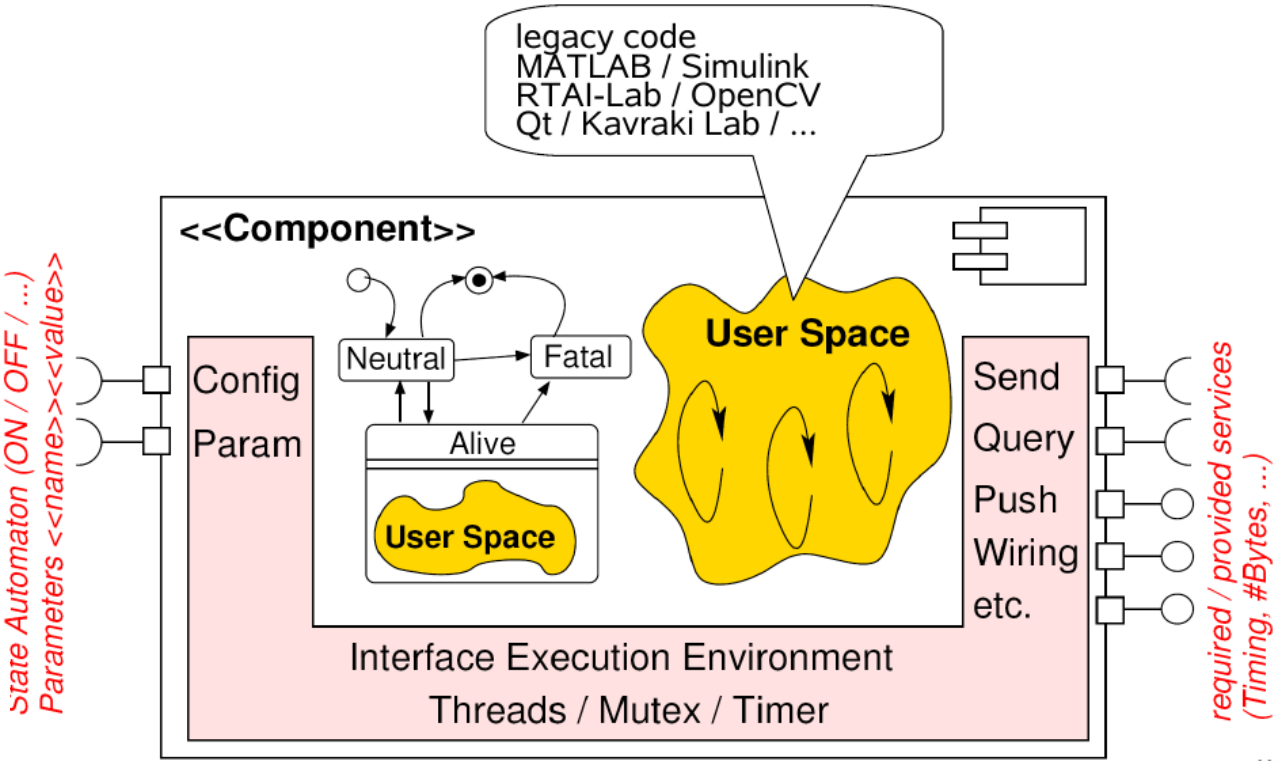
# Model Driven Software Development
## Idea and Approach

```
#include "smartSoft.hh"
#include "commVoid.hh"
#include "commMobileLaserScan.hh"

CHS::QueryClient<CommVoid,CommMobileLaserScan> *laserQueryClient;

// separate thread for user activity
class UserThread : public CHS::SmartTask {
public:
    UserThread() {};
    ~UserThread();
    int svc(void);
};

int UserThread::svc(void) {
    CommVoid request1, request2;
    CommMobileLaserScan answer1, answer2;
    CHS::QueryId id1, id2;
    ...
    status = laserQueryClient->request(request1,id1);
    status = laserQueryClient->request(request2,id2);
    ...
    status = laserQueryClient->receiveWait(id2,answer2);
    status = laserQueryClient->receiveWait(id1,answer1);
    ...
}
...
int main(int argc,char *argv[]) {
    ...
    CHS::SmartComponent component("first",argc,argv);
    CHS::WiringSlave wiring(component);
    UserThread user;

    laserQueryClient = new CHS::QueryClient<CommVoid,CommMobileLaser>
                    (component,"laserPort",wiring);
    user.open();
    component.run()
    ...
}
```

```
#include "smartSoft.hh"
#include "commVoid.hh"
#include "commMobileLaserScan.hh"

// this handler is executed with every incoming query
class LaserQueryHandler
    : public CHS::QueryServerHandler<CommVoid,CommMobileLaserScan> {
public:
    void handleQuery(
            CHS::QueryServer<CommVoid,CommMobileLaserScan>& server,
            const CHS::QueryId id,
            const CommVoid& r) throw()
    {
        CommMobileLaserScan a;
        // request r is empty in this example, now calculate an answer
        server.answer(id,a);
    }
};

int main(int argc,char *argv[])
{
    ...
    // the component management is mandatory in all components
    CHS::SmartComponent component("second",argc,argv);
    // the following implements a query service for laser scans
    // with an active handler
    LaserQueryHandler laserHandler;
    CHS::QueueQueryHandler<CommVoid,CommMobileLaserScan>
                    activeLaserHandler(laserHandler);
    CHS::QueryServer<CommVoid,CommMobileLaserScan>
                    laserServant(component,"laser",activeLaserHandler);
    ...
    // the following call operates the framework by the main thread
    component.run();
}
```

# Model Driven Software Development
## Idea and Approach






Hochschule Ulm

# Model Driven Software Development
# What do we need for a component model ?

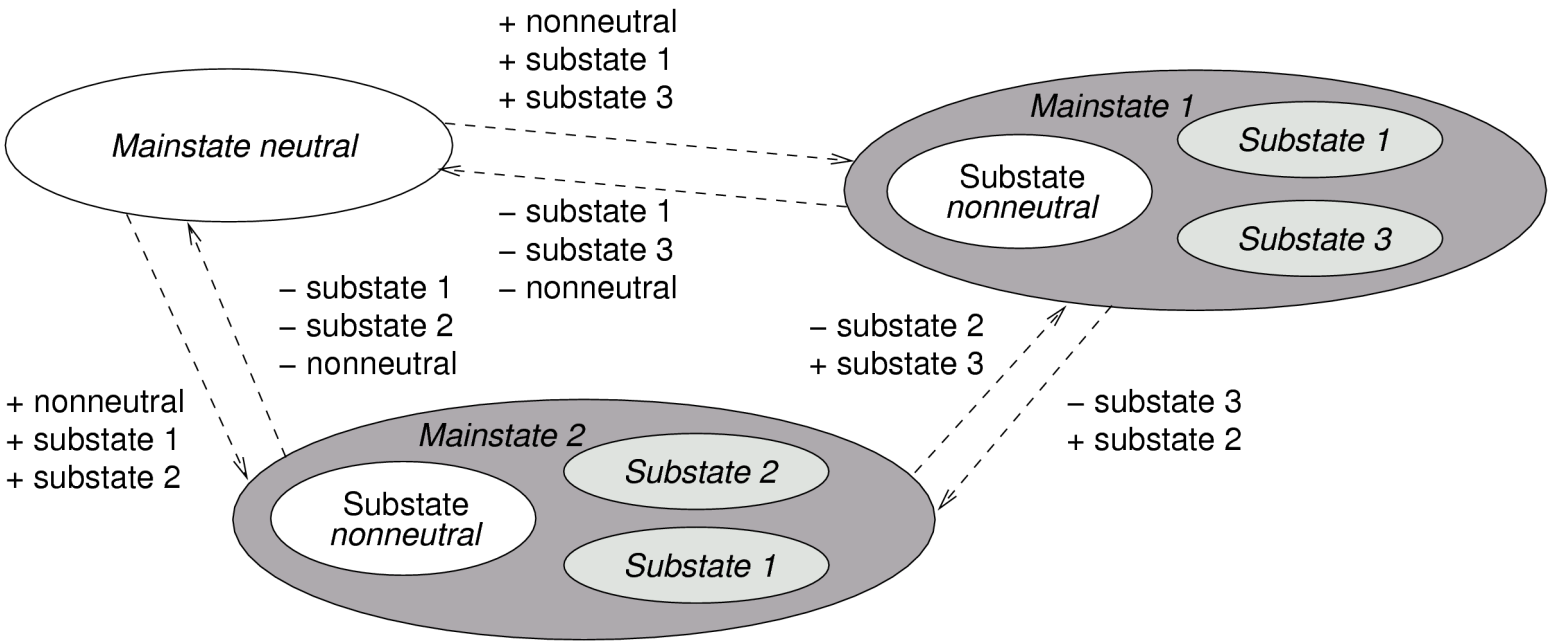
- Interaction Patterns
  - loosely coupled communication
  - independent of middleware
  - accessible to MDSD
- Parameterization and configuration ports
  - name / value pairs
  - dynamic wiring
  - reflection ?
- Abstraction of execution container
  - resource access via abstraction independent of implementational technology and OS
  - Tasks, Semaphore, CV, PCP, etc.
  - accessible to MDSD
- State automaton inside a component
  - share common states to support orchestration
  - allow for individual substates beneath basic state automaton
- and many others
  - authorization, encryption, etc. etc.

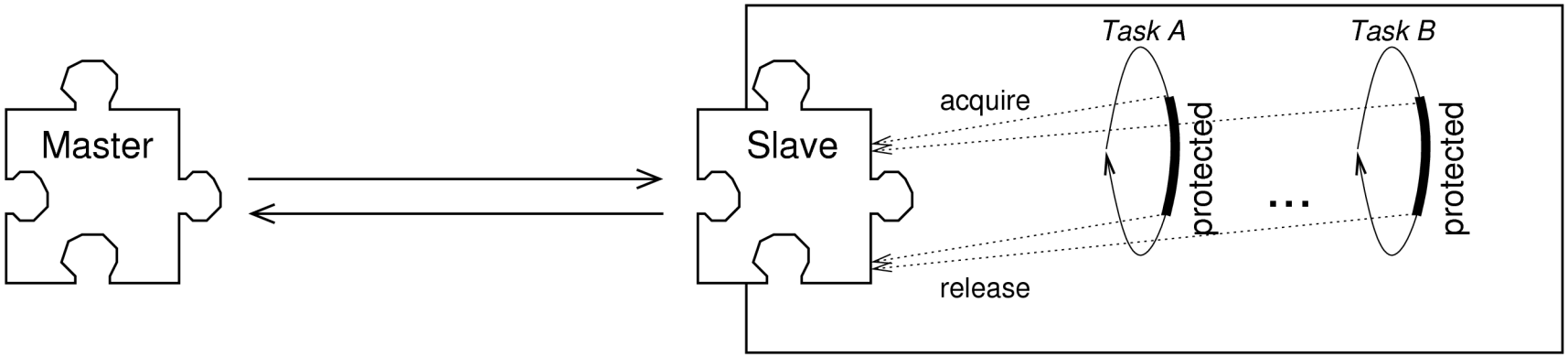
# Model Driven Software Development
## Idea and Approach

| **Wiring Master** |
| --- |
| |
| + WiringMaster(:SmartComponent*) throw(SmartError)<br>+ *~WiringMaster() throw() [virtual]*<br><br>+ blocking(flag:const bool) : StatusCode throw()<br><br>+ connect(slavecmpt:const string&, slaveprt:const string&, servercmpt:const string&, serversvc:const string&) : StatusCode throw()<br>+ disconnect(slavecmpt:const string&, slaveprt:const string&) : StatusCode throw() |

| **Wiring Slave** |
| --- |
| |
| + WiringSlave(:SmartComponent*) throw(SmartError)<br>+ *~WiringSlave() throw() [virtual]* |

# Model Driven Software Development
## Idea and Approach



**State Master**

+ StateMaster(:SmartComponent*) throw(SmartError)
+ StateMaster(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ StateMaster(:SmartComponent*, port:const string&, :WiringSlave*) throw(SmartError)
+ *~StateMaster() throw() [virtual]*

+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()

+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()

+ blocking(flag:const bool) : StatusCode throw()

+ setStateWait(state:const string&) : StatusCode throw()
+ getCurrentStateWait(state:string&) : StatusCode throw()
+ getMainStatesWait(states:list<string>&) : StatusCode throw()
+ getSubStatesWait(mainstate:const string&,states:list<string>&) : StatusCode throw()

---

**State Slave**

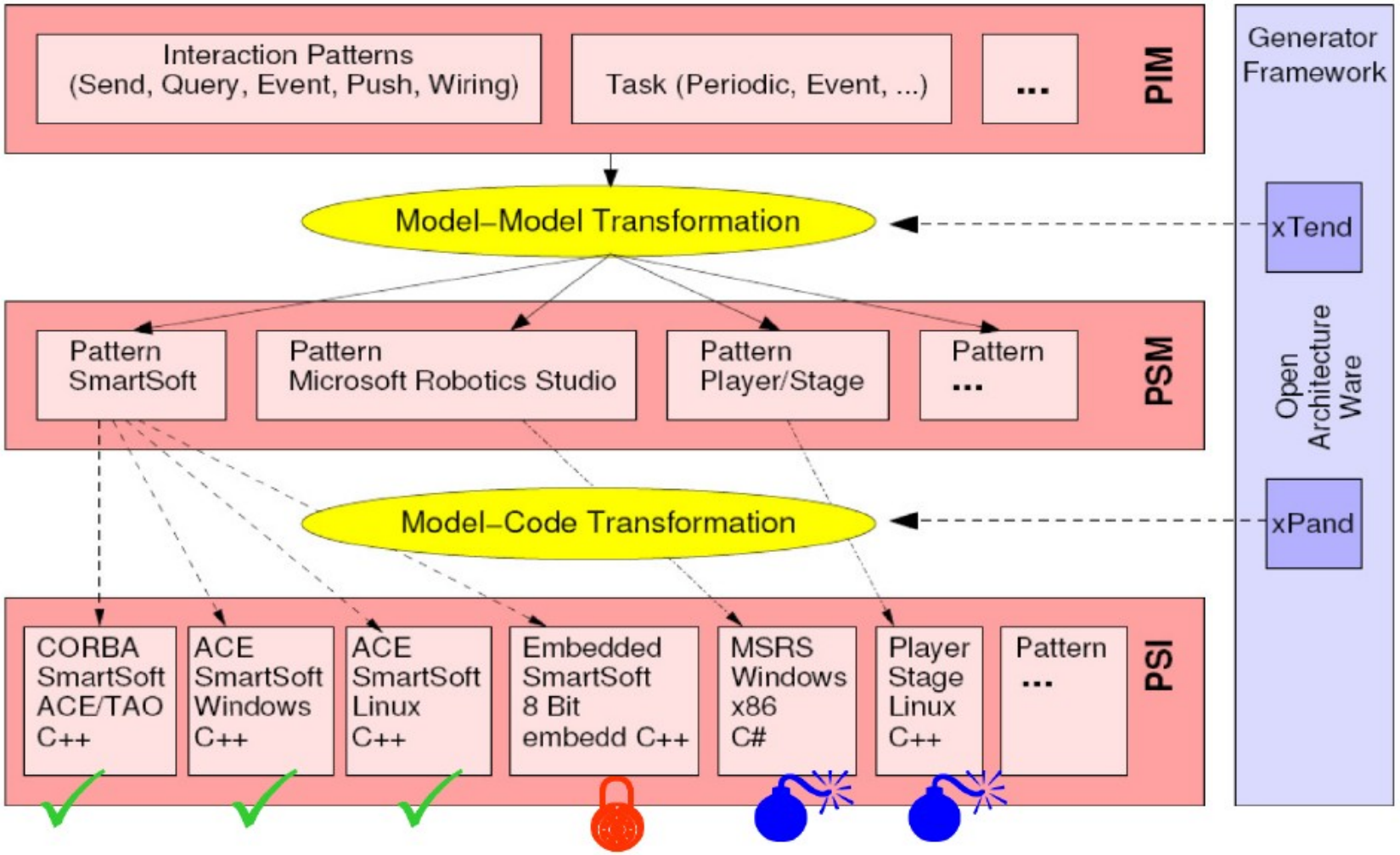
+ StateSlave(:SmartComponent*, service:const string&, :StateChangeHandler&) throw(SmartError)
+ *~StateSlave() throw() [virtual]*

+ defineStates(mainstate:const string&, substate:const string&) : StatusCode throw()
+ activate() : StatusCode throw()
+ acquire(substate:const string&) : StatusCode throw()
+ tryAcquire(substate:const string&) : StatusCode throw()
+ release(substate:const string&) : StatusCode throw()

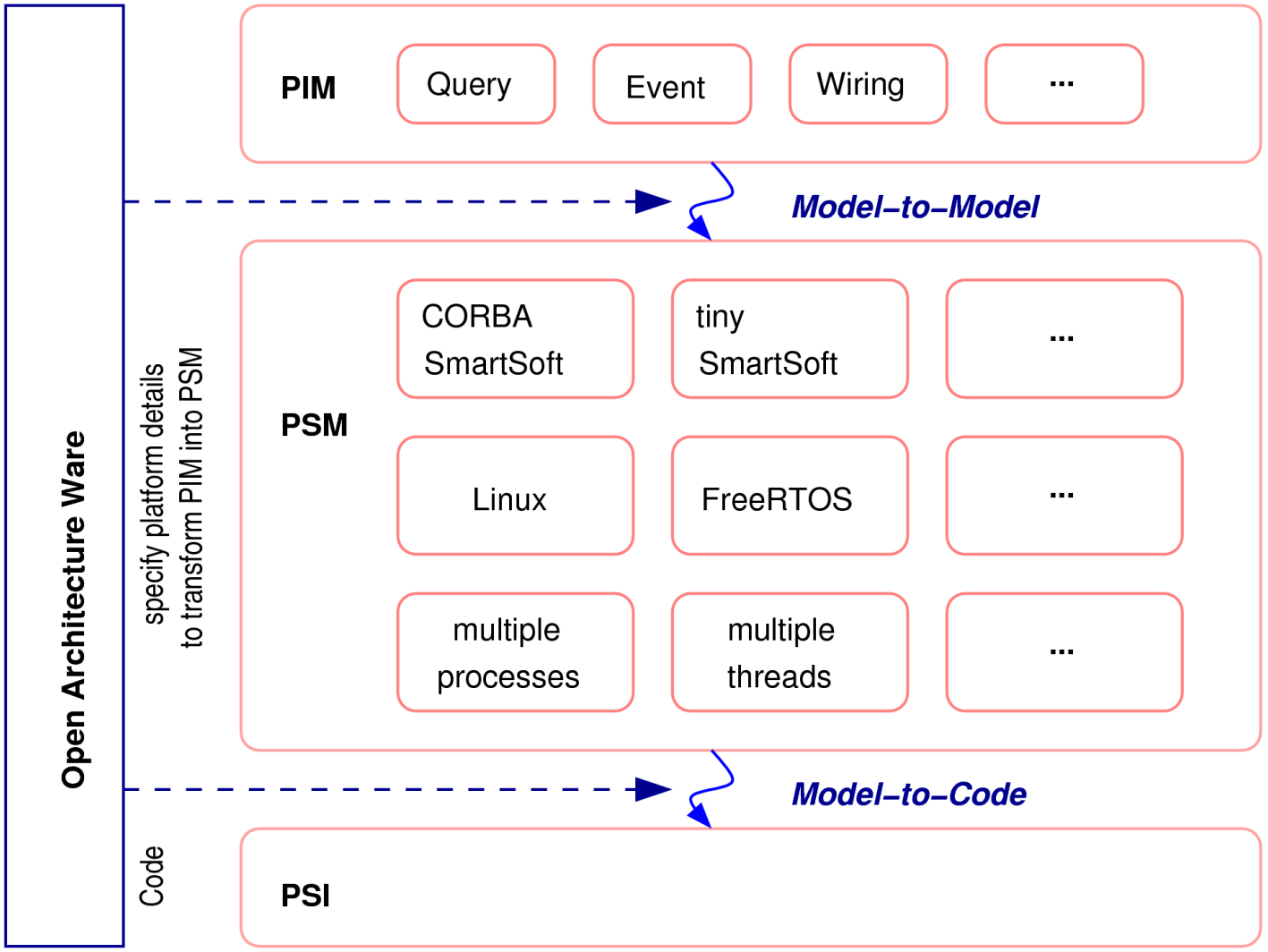
***State Change Handler {abstract}***

+ *handleEnterState(state:const string&) : void throw() [pure virtual]*
+ *handleQuitState(state:const string&) : void throw() [pure virtual]*

# Model Driven Software Development
# Idea and Approach

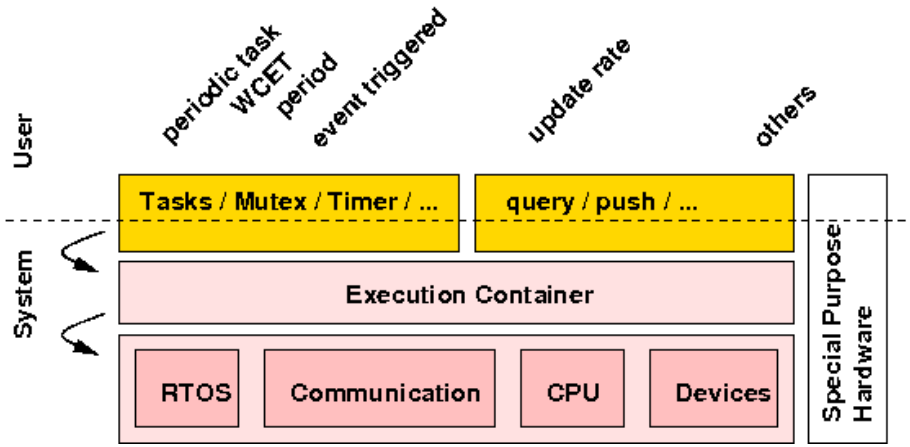


# Model Driven Software Development
## Idea and Approach



# Model Driven Software Development
# Mapping onto Different Platforms

- Communication Objects
  - marshalling
- Communication Patterns
  - message system beneath patterns (downward interface)
    - ACE Reactor / Acceptor etc.
    - CORBA
    - 0NQ message system
    - global variables
  - no adjustment towards user (upward interface)
- Tasks etc.
  - obvious, on no-realtime systems just no guarantee for periods etc.

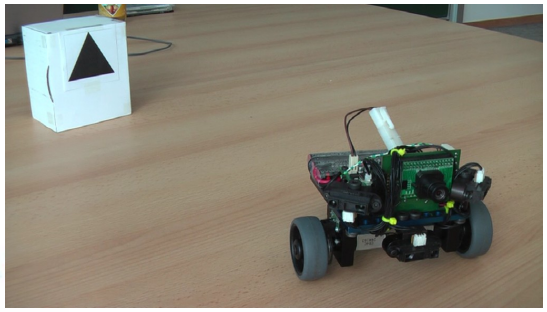


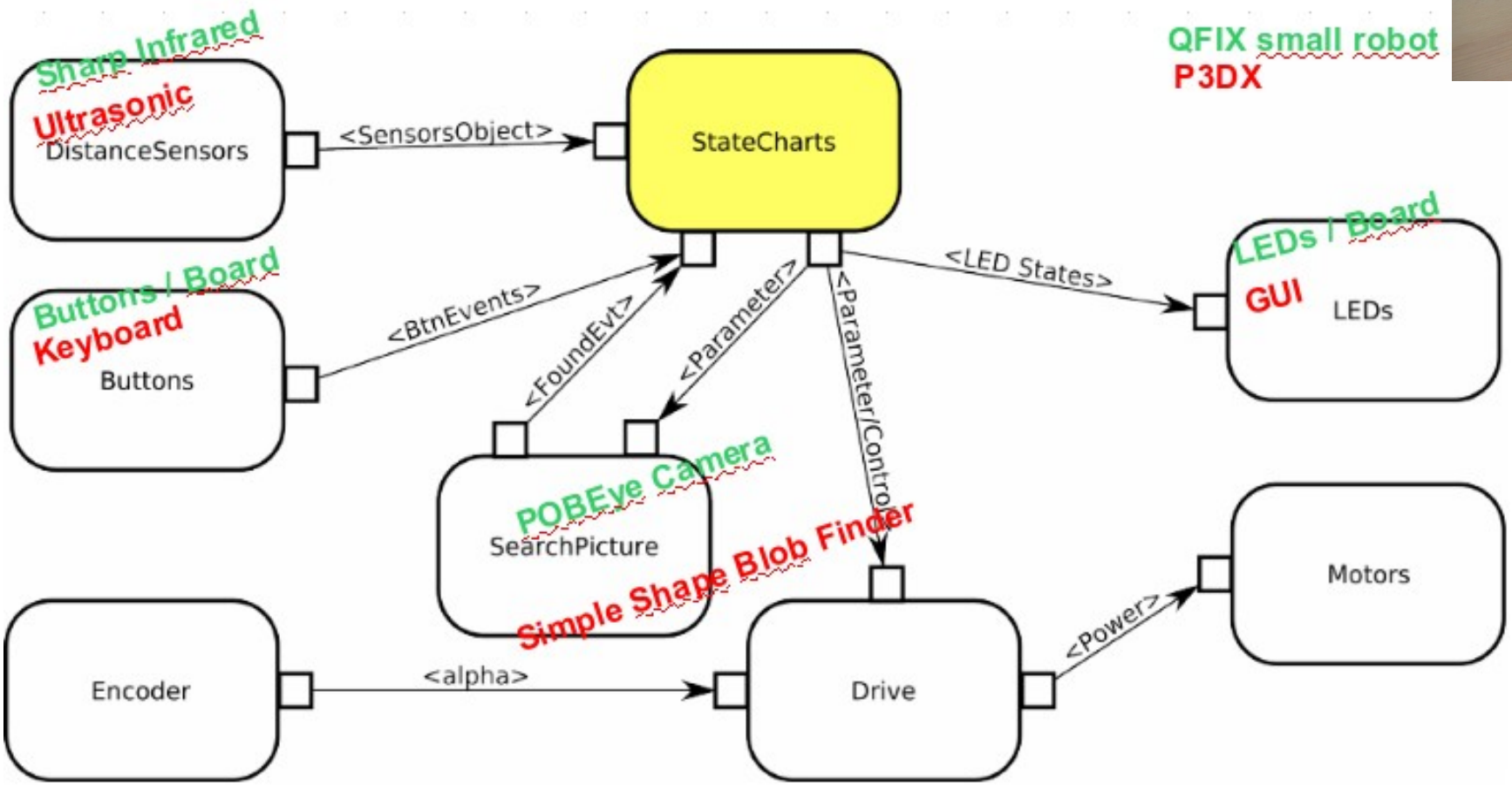
# Model Driven Software Development
# Extensions towards QoS / real-time

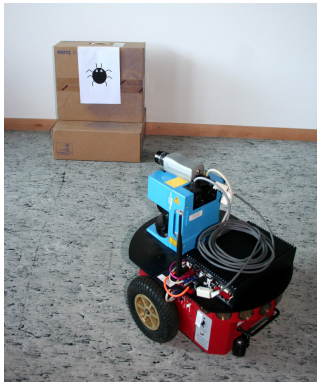
- already possible
  - have timing parameters within communication objects as part of request to server
  - server can then response with void answer in case it cannot meet the deadline
- next step
  - timeout-parameters at user-interface of interaction methods
    - 0 infinity / no timeout
    - others timeout
    - *since interaction patterns are standalone entities, these timings are easily implemented locally without server interaction (see priniciple of locality)*
    - *have these parameters within UML component model*
- next step
  - deployment of components
    - map ports (and their messages) onto hard real-time communication systems like Realtime-Ethernet
    - extend UML port model with period / load etc. for RMA scheduling analysis
- next step
  - extend this towards general resource awareness
    - tasks / RMA dependent on PSM
    - interfaces to external tools like SuReal / AbsInt etc.

service robotik

# Model Driven Software Development Example



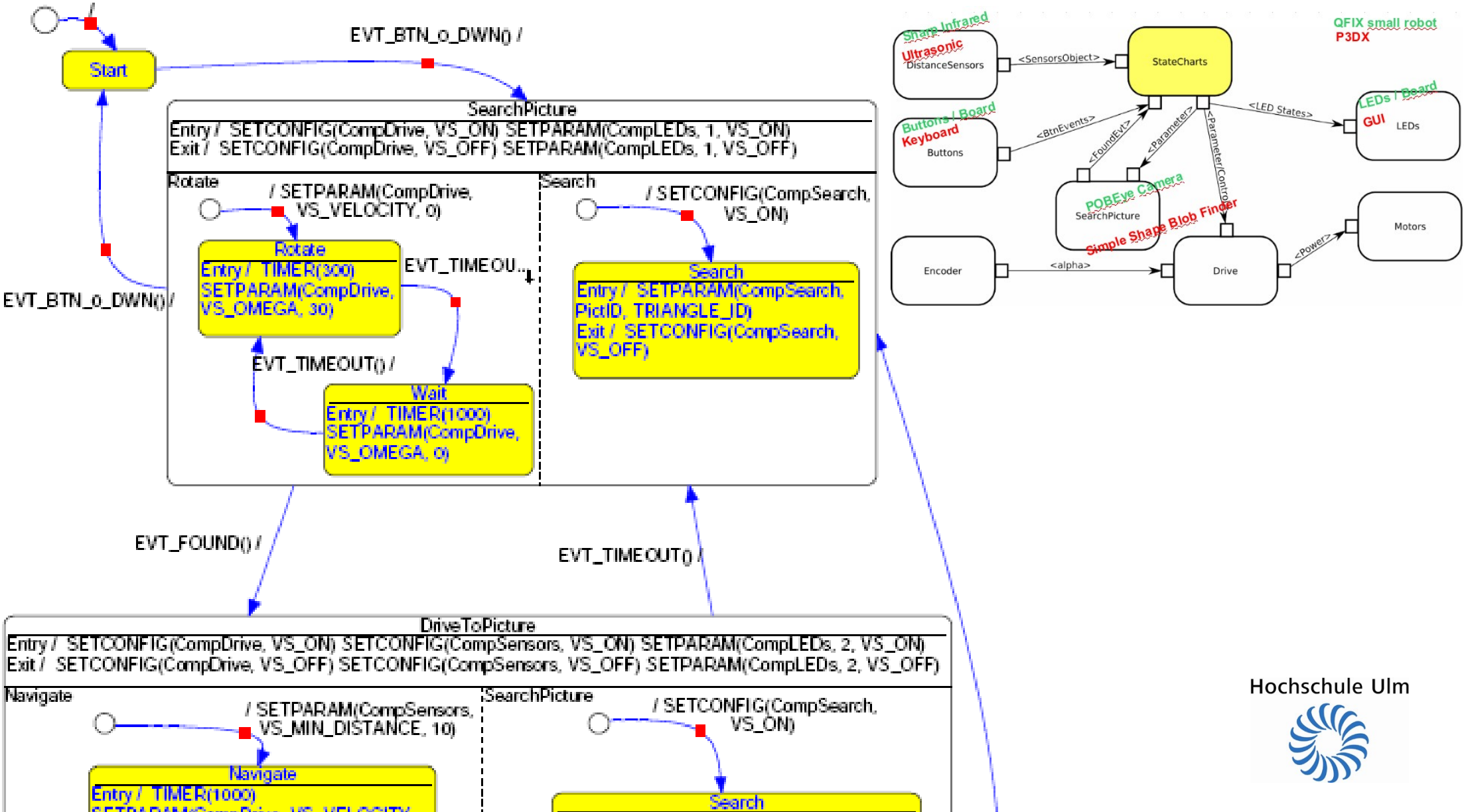


Hochschule Ulm

service robotik

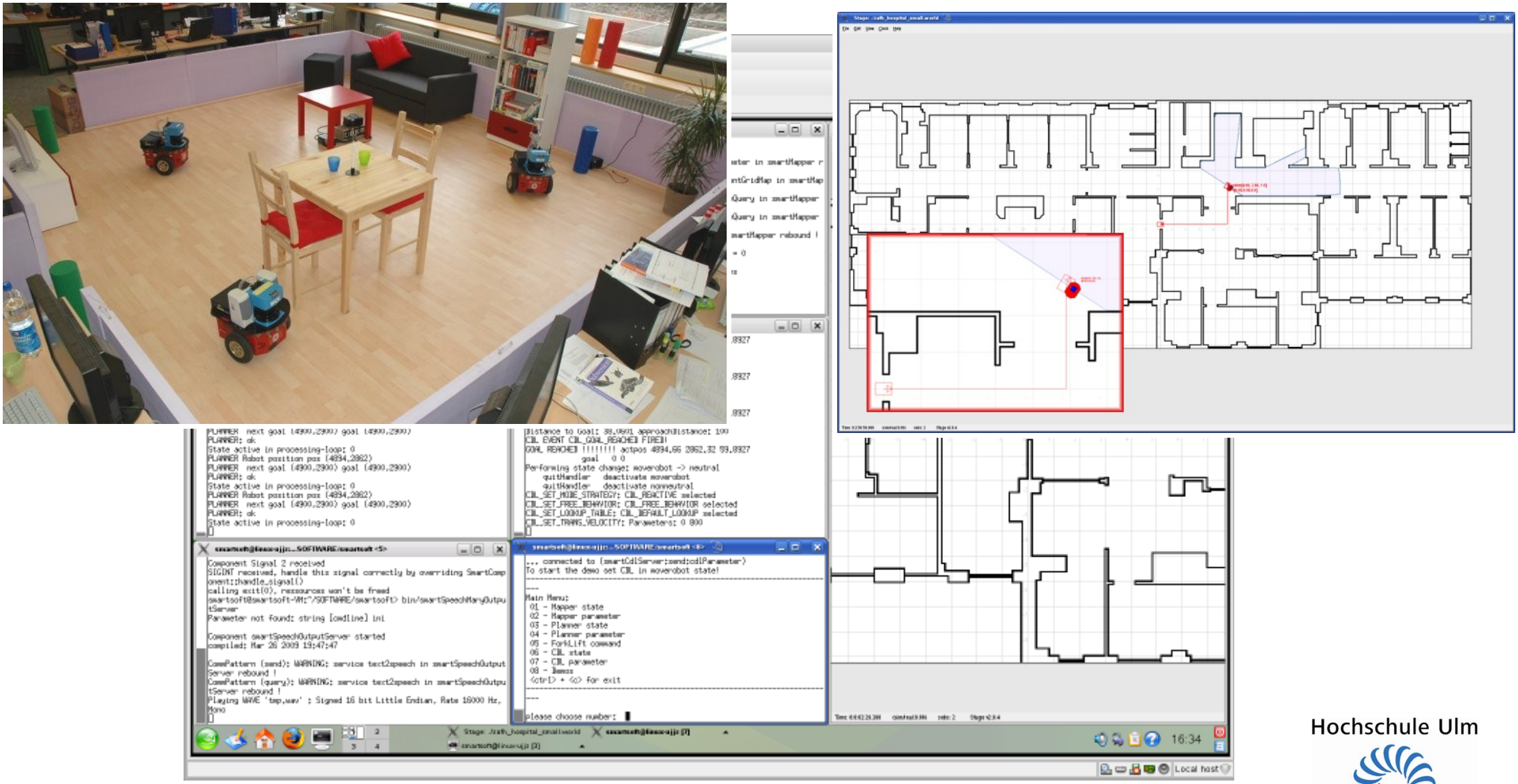
# Model Driven Software Development Example




Hochschule Ulm

service robotik

# Model Driven Software Development
# Summary and Conclusion






Hochschule Ulm