Model Driven Software Development in Robotics – *It really works*!

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
http://smart-robotics.sourceforge.net/

Siegfried Hochdorfer, Alex Lotz, Matthias Lutz, Dennis Stampfer, Andreas Steck, Jonas Brich, Manuel Wopfner
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, standards etc. also useful in robotics!
A development process often applied in robotics ...
Systematic engineering processes look different ... 
... also in case of software intensive systems
We need a systematic engineering approach for robotics software!

- robots are complex systems that depend on systematic engineering
- so far fundamental properties of robotic systems have not been made detailed enough nor explicit (e.g. QoS)
- tremendous code-bases (libraries, middleware, etc.) coexist without any chance of interoperability and each tool has attributes that favors its use

→ rely, as for every engineering endeavour, on the power of models
→ nowadays, robotics functionality is foremost based on software
→ make the step towards MDSD
What is Model Driven Software Development?

- make software development more domain related as opposed to computing related
- it is also about making software development in a certain domain more efficient and more robust due to design abstraction

http://www.voelter.de/services/mdsd.html
Modelling + Formalization = Solution of all Problems?

- “the earlier the formalization, the more steps can be automated“ => is it true?
- what is about software architecture and target platform?
  - need also be available in formalized form to be accessible by transformations!
  - that is exactly what is required by MDA in form of PIM and PSM
  - both, the software architecture as well as the platforms are formally described by models
How MDSD works

- Developer develops model(s) based on certain metamodel(s), expressed using a DSL
- Using code generation templates, the model is transformed to executable code - alternative: interpretation
- Optionally, the generated code is merged with manually written code
- One or more model-to-model transformation steps may precede code generation

http://www.voelter.de/services/mdsd.html
Why is Model Driven Software Development important in robotics?

- Software development is too complex and too expensive

... because:

- there is too little reuse
- technology changes faster than developers can learn
- knowledge and practices are hardly captured explicitly and made available for reuse
- domain experts cannot understand all the technology stuff involved in software development
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 transformers
- 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
Model Driven Software Development
Example / Navigation Task
What is different in robotics?

- **not** the huge number of different sensors and actuators
- **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 adequately solve tasks given certain resources
That sounds good but give me an example ...

we made some very simple but pivotal decisions while dealing with component based systems that then proved to pave the way towards MDSD:

- 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  
  \[\text{independent of a certain middleware}\]  
  \[\text{enforce standardized service contracts between components}\]

- minimum component model to support system integration  
  - dynamic wiring of the data flow between components  
  - state automaton to allow for orchestration / configuration  
  \[\text{ensures composability / system integration}\]
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 as 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 it rejects requested response time
    - 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
Idea and Approach

<table>
<thead>
<tr>
<th>send</th>
<th>CHS::SendClient, CHS::SendServer, CHS::SendServerHandler</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>CHS::QueryClient, CHS::QueryServer, CHS::QueryServerHandler</td>
</tr>
<tr>
<td>push newest</td>
<td>CHS::PushNewestClient, CHS::PushNewestServer</td>
</tr>
<tr>
<td>push timed</td>
<td>CHS::PushTimedClient, CHS::PushTimedServer, CHS::PushTimedHandler</td>
</tr>
<tr>
<td>event</td>
<td>CHS::EventClient, CHS::EventHandler, CHS::EventServer, CHS::EventTestHandler</td>
</tr>
<tr>
<td>wiring</td>
<td>CHS::WiringMaster, CHS::WiringSlave</td>
</tr>
</tbody>
</table>
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()
- Communication Objects
  - marshalling
- Communication Patterns
  - downward interface / internal
    - invisible to user
    - is handled by MDSD toolchain
    - can be mapped onto different middleware systems
      » ACE Reactor / Acceptor etc.
      » CORBA
      » 0MQ message system
      » global variables
  - upward interface / user
    - no adjustments at user visible API
- Tasks etc.
  - obviously, no guarantees when mapped onto no-realtime systems etc.
Model Driven Software Development
SmartMDSD

Open Architecture Ware
- PIM: Query, Event, Wiring, ...
- PSM: CORBA, SmartSoft, tiny SmartSoft, ...
- Code: Linux, FreeRTOS, multiple processes, multiple threads, ...
- PSI

Model-to-Model
Model-to-Code

Increased abstraction
Expansion / translation towards code
Benefits of this development process:

- systematically handle integration of systems of the complexity of service robots and to overcome plumbing
- tools like OpenArchitectureWare, Eclipse etc. are matured enough to be used in robotics
- there are many experienced people out there being already familiar with the tools, can start immediately using them and can just focus on robotics
- design patterns, best practices, approved solutions can be made available within the code generators such that even novices can immediately take advantage from that coded and immense experience
- SmartSoft provides the perfect granularity for system design, component development, composability, freedom within components, tool support etc.
Model Driven Software Development
/ MDSD basics /

MDSD core building blocks:
- domain analysis
- meta modelling
- model-driven generation (and: model transformations, model-to-model, model-to-text)
- template languages
- domain-driven framework design

Are MDSD models the same as requirements / analysis models?
- they can be, but in general, they are not
- analysis / requirements models are non-computational, MDSD models are computational
- formalizing requirements is beneficial since requirements become unambiguous
- MDSD models are no “paperwork“, they are the solution which is translated into code automatically

http://www.voelter.de/services/mdsd.html
Three basic viewpoints:

- **Type Model**: Components, Interfaces, Data Types
- **Composition Model**: Instances, „Wirings“
- **System Model**: Nodes, Channels, Deployments

Generated stuff:

- base classes for component implementation
- build scripts
- descriptors
- remoting infrastructure
- persistence
- etc.

http://www.voelter.de/services/mdsd.html
Aspect models:

- often, the described three viewpoints are not enough, **additional aspects** need to be described
- these go into **separate aspect models**, each describing a well-defined aspect of the system
  - each of them uses a suitable DSL / syntax
  - the generator acts as a weaver
- Typical **examples** are
  - persistence
  - security
  - timing, QoS in general
  - packaging and deployment
  - diagnostics and monitoring

http://www.voelter.de/services/mdsd.html
Illustration of our development process

- UML 2.0 profile for robotics component model
- covers component development, system composition, deployment
- based on standards: UML 2.0, Open Architecture Ware, Eclipse, etc.
- different runtime platforms, middleware systems etc.

SmartMARS (Modeling and Analysis of Robotics Systems)
M2M oAW xTend
M2T oAW xPand
CorbaSmartSoft CORBA based implementation of SmartSoft
AceSmartSoft ACE based implementation of SmartSoft

User Code
legacy code
MATLAB / Simulink
RTAI-Lab
OpenCV / Qt / OpenRave

User Space
Config
Param
Alive
Neutral
Fatal
Threads / Mutex / Timer
Interface Execution Environment

Send
Query
Push
Wiring
etc.
Model Driven Software Development
SmartMDSD

SmartMARS – Metamodel
(Modeling and Analysis of Robotics Systems)

PIM

- UML2-Profile
- platform independent stereotypes
  - SmartComponent
  - SmartTask
  - SmartMutex
  - SmartQueryServer
  - SmartEventClient
  - ...

PSM

- UML2-Profile
- platform specific stereotypes
  - AceSmartSoft
    - ACE based implementation of SmartSoft
  - CorbaSmartSoft
    - CORBA based implementation of SmartSoft
  - ... any other middleware

PSI

The user space can contain arbitrary code and libraries
The user space remains the same independent of the different platform specific models
Just the component hull will be created

CorbaSmartSoft
CORBA based implementation of SmartSoft

AceSmartSoft
ACE based implementation of SmartSoft

AceSmartSoft
ACE based implementation of SmartSoft

M2M xTend check

PSI has to be created by a middleware expert

M2T xPand check

Send Query
Push Wiring etc.

Hochschule Ulm

7 May 2010
SDIR V / Schlegel, Steck
What do we need within a component meta-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.
Model Driven Software Development

Metamodels (partial view)
Model Driven Software Development
SmartMDSD
Model Driven Software Development
Examples / SmartMDSD / Real-Time Task

<<enumeration>>
SchedPolicyKind
FIFO
round-robin
sporadic

<<metaelement>>
SmartTask
schedPolicy: SchedPolicyKind
isRealtime: Boolean
isPeriodic: Boolean
priority: Integer
timeUnit: TimeUnitType
period: Integer
wcet: Integer

isRealtime == false

timer [0..1]
isPeriodic == true

<<metaelement>>
SmartCorbaTask
schedPolicy: SchedPolicyKind
isPeriodic: Boolean
priority: Integer
period: Integer

<<metaelement>>
SmartCorbaCondMutex
condMutex [0..1]

<<metaelement>>
SmartCorbaTimer
period: Integer

<<metaelement>>
RTAITask
schedPolicy: SchedPolicyKind
isPeriodic: Boolean
priority: Integer
period: Integer
wcet: Integer

<<metaelement>>
SmartCorbaMutex
mutex [1]
already available
- 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
- interface to CHEDDAR timing analysis for RMA / dependent on PSM

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 principle of locality)
  – have these parameters within UML component model

next step
- at deployment of components
  – map ports (and their messages) onto hard real-time communication systems where needed (like Realtime-Ethernet)
- extend this towards general resource awareness

incrementally extend Meta-Models to cover more and more aspects of robotics
Model Driven Software Development
Examples etc.
Toolchain based on Open Architecture Ware
- fully integrated into Eclipse
- [http://www.openarchitectureware.org/](http://www.openarchitectureware.org/)

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]
- CORBA (ACE/TAO) based SmartSoft
  - on sourceforge with various robotics components and simulators etc.
  - in use in research and industry
- ACE (without CORBA) based SmartSoft
  - on sourceforge [Linux, Windows]
  - in use in research and industry
- oAW Toolchain for SmartSoft
  - on sourceforge (including Screencasts and Tutorials)
SmartSoft MDSD Toolchain

CORBA / SmartSoft

ACE / TAO
- Linux
- RTAI

ACE / SmartSoft

ACE
- Linux
- Windows
- QNX

Linux
- RTAI

Eclipse

PapyrusUML

oAW

under construction

>> more

>> more
Addendum
## Query Client

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>QueryClient(:SmartComponent*)</code></td>
<td><code>throw(SmartError)</code></td>
</tr>
<tr>
<td><code>QueryClient(:SmartComponent*, server:const string&amp;, service:const string&amp;)</code></td>
<td><code>throw(SmartError)</code></td>
</tr>
<tr>
<td><code>QueryClient(:SmartComponent*, port:const string&amp;, slave:WiringSlave*)</code></td>
<td><code>throw(SmartError)</code></td>
</tr>
<tr>
<td><code>~QueryClient()</code></td>
<td><code>throw()</code> [virtual]</td>
</tr>
<tr>
<td><code>add(:WiringSlave*, port:const string&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>remove()</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>connect(server:const string&amp;, service:const string&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>disconnect()</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>blocking(flag:const bool)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>query(request:const R&amp;, answer:A&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>queryRequest(request:const R&amp;, id:QueryId&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>queryReceive(id:const QueryId, answer:A&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>queryReceiveWait(id:const QueryId, answer:A&amp;)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
<tr>
<td><code>queryDiscard(id:const QueryId)</code></td>
<td><code>: StatusCode throw()</code></td>
</tr>
</tbody>
</table>
Model Driven Software Development
/ Technical Details /

**Query Server**

+ QueryServer(:SmartComponent*, service:const string&, :QueryServerHandler<R,A>&) throw(SmartError)
+ ~QueryServer() throw() [virtual]

+ StatusCode answer(:const QueryId, answer:const A&) throw()
+ StatusCode check(:const QueryId) throw()
+ StatusCode discard(:const QueryId) throw()

**Query Server Handler {abstract}**

+ handleQuery(server:QueryServer<R,A>&, id:const QueryId, request:const R&) : void throw() [pure virtual]

**Queue Query Server Handler {active}**
### Wiring Master

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiringMaster(SmartComponent*)</td>
<td>throw(SmartError)</td>
</tr>
<tr>
<td>~WiringMaster()</td>
<td>throw() [virtual]</td>
</tr>
<tr>
<td>blocking(flag:const bool)</td>
<td>: StatusCode throw()</td>
</tr>
<tr>
<td>connect(slavecmt:const string&amp;, slaveprt:const string&amp;, servercmt:const string&amp;, serversvc:const string&amp;)</td>
<td>: StatusCode throw()</td>
</tr>
<tr>
<td>disconnect(slavecmt:const string&amp;, slaveprt:const string&amp;)</td>
<td>: StatusCode throw()</td>
</tr>
</tbody>
</table>

### Wiring Slave

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WiringSlave(SmartComponent*)</td>
<td>throw(SmartError)</td>
</tr>
<tr>
<td>~WiringSlave()</td>
<td>throw() [virtual]</td>
</tr>
</tbody>
</table>
### State Master

```cpp
+ 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]
```

```cpp
+ 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:const string&) : StatusCode throw()
+ getMainStatesWait(states: list<string>&) : StatusCode throw()
+ getSubStatesWait(mainstate:const string&, states: list<string>&) : StatusCode throw()
```

### State Slave

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

```cpp
+ 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}

```cpp
+ handleEnterState(state:const string&) : void throw() [pure virtual]
+ handleQuitState(state:const string&) : void throw() [pure virtual]
```
Model Driven Software Development Example
Model Driven Software Development

Idea and Approach