MDD for Embedded Systems

Markus Völter
voelter@acm.org
www.voelter.de

About me

- Independent Consultant
- Based out of Heidenheim, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Middleware
CONTENTS

• What is MDSD
• What are Embedded Systems
• The Challenges and MDD
  • Resource Efficiency
  • Architectural Consistency
  • System Complexity
  • Product Lines & Variabilities
  • Platform Independence / Portability
  • Tool Integration
  • Hardware & Systems Engineering
• Other Relevant Topics
  • Model-to-Model Transformations
  • Variability Management and Configuration
  • Standards
• Summary
Model-Driven Software Development

- MDSD is about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.

![Diagram showing Domain Concepts and Software Technology Concepts]

**MDSD on a Slide**

- Several design-expertise
- Multi-step single-step
- No round-trip
- Several steps
- Transform compile interpret
- Precise executable
- Domain-specific language
- Bounded area of knowledge/interest
- Metamodel
- Subdomain
- Partial viewpoint
- Multi-viewpoint
- Graphical textual
- Domain Ontology
- Subdomain
How does MDSD work?

- Developer develops **model(s)** based on certain metamodel(s).
- Using **code generation templates**, the model is transformed to executable code.
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.

### CONTENTS

- What is MDSD
- **What are Embedded Systems**
- The Challenges and MDD
  - Resource Efficiency
  - Architectural Consistency
  - System Complexity
  - Product Lines & Variabilities
  - Platform Independence / Portability
  - Tool Integration
  - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary
What are Embedded Systems

- Well, a very diverse crowd 😊

Characteristics of Embedded Systems

- **Non-homogeneity**, almost every device is different
- **Limited memory**, often only some hundred kB or few MB are available
- **Limited computing power** because the devices often contain only small processors
- **Limited electrical power** because the device might be battery powered
- **Timing requirements** can be more stringent:
  - Soft real time
  - Hard real time
- **Network connection** can either be permanent or intermittent
- **High system complexity**, “configuration hell”
- **Concurrency**, simulated (threads, processes) and real (many nodes in the system)
- **Hardware-Dependent**, integration with various sensors and communication technologies
Challenges

- To develop **resource efficient** systems with as little runtime overhead as possible
- To build **architecturally coherent** systems
- To handle **system complexity** including **QoS aspects**
- To systematically handle the **variabilities** among different products in a product line
- Often they need to run on **different OS / hardware** configurations
- To **integrate the various tools** used in the production process
- Consider the **hardware** ("Systems Engineering")

**CONTENTS**

- What is MDSD
- What are Embedded Systems
- The Challenges and MDD
  - **Resource Efficiency**
  - Architectural Consistency
  - System Complexity
  - Product Lines & Variabilities
  - Platform Independence / Portability
  - Tool Integration
  - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary
Context

- In embedded systems, indirection, frameworks and other **classical means of expressing abstractions** cannot be used for reasons of performance and other determinism.

- With MDSD, **abstractions can still be expressed** at modeling/source time. Using code generation, an efficient **implementation can be generated** that does not suffer from any performance overhead.

- Code generators can include **optimizations** (just as compilers do)
  - to generate the **most efficient implementation** wrt. to certain properties (space, time, ...)
  - **Parts/aspects** of the specific system instance (think product lines) can be **ommitted** from the generated system

What is efficient code?

- For example, in C you can do the following:
  - Deliberately making things **inline**
  - Using **function pointers** (maybe hidden behind macros) to access configurable functionality
  - Decision tables based on **arrays indexing into each other** (e.g. for state machines)
  - Generating code to directly call a (potentially remote) entity, thereby “removing the middleware” completely
  - Based on application features, generate the **smallest possible OS configuration**

- Manually coding some of these things is very **tedious and error prone**
  - This is why things are **brittle** when used in manual coding
  - But a **generator** can use these techniques **consistently and reliably**
CONTENTS

- What is MDSD
- What are Embedded Systems
- The Challenges and MDD
  - Resource Efficiency
  - Architectural Consistency
  - System Complexity
  - Product Lines & Variabilities
  - Platform Independence / Portability
  - Tool Integration
  - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary

Context

- Many embedded systems are actually large, distributed networked systems where each part must play along certain rules to make the overall system work.
  - Often, many different organizations/vendors cooperate to engineer the overall system (e.g. automotive systems)

- A common architecture must be established and enforced to make the system deliver on its (functional and non-functional) requirements

- MDSD can help by
  - Formally defining architectures via architectural meta models
  - Checking constraints on actual applications models (static, as well as dynamic)
  - Generating implementation skeletons for developers to add their application logic in an architecturally conforming way
Example: A CBD Meta Model (Type Viewpoint)

- Components
- Interfaces
- Operations

Example: A CBD Meta Model (Composition Viewpoint)

- Component Instances
- Connectors, "Wiring"
Example: A CBD Meta Model (System Viewpoint)

- Hardware (Processors, Networks, Busses)
- Deployment & Configuration

Example: A CBD Meta Model (Constraints)

- A wire’s target’s type must provide the interface to which the wire’s interface requirements points:

  ```
  context Wire ERROR "wrong type at the other end":
  target.type.providedPorts.interface.exists( i | i == cireq.target );
  ```

- The required interface objects of a component need to have unique names (that’s an assumption required by the code generator)

  ```
  context Component ERROR "ports must have unique names":
  requiredInterface.forAll( r1 |
  !requiredInterface.exists( r2 |
  r1 != r2 && r1.name == r2.name ) );
  ```
Example: AUTOSAR

- AUTOSAR is a **standard software architecture for automotive ECUs**
  - although large portions could be used outside automotive systems, too.

- It addresses many aspects of automotive software, the most important aspects in my opinion are
  - Formal definition of what a **component** is
  - A **deployment, configuration and communications infrastructure** for component instances
  - A **modeling language** (based on a formal meta model) for describing AUTOSAR systems

**CONTENTS**

- What is MDSD
- What are Embedded Systems
- The Challenges and MDD
  - Resource Efficiency
  - Architectural Consistency
  - **System Complexity**
    - Product Lines & Variabilities
    - Platform Independence / Portability
    - Tool Integration
    - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary
Context

- Distributed Realtime-Embedded (DRE) systems are among the **most complex systems on the planet**: large, distributed, QoS-aware, reliable.

- These systems consist of so many parts that describing, managing and orchestrating them requires sophisticated abstractions and formalisms.

- The systems built from these parts need to **address certain QoS properties** (i.e. global constraints) wrt. to timing, resource consumption, liveliness, etc.

- MDSD can help by
  - Formally describing the parts that make up the system including their metadata necessary for QoS
  - From the models, simulations can be derived that check whether certain dynamic properties hold
  - Various algorithms (constraint solvers, etc.) can help in coming up with (more or less) ideal system configurations for meeting the global constraints.

The Role of Patterns

- Patterns are **essential** in documenting best practices ("proven solutions for recurring problems").

- Patterns document considerations, context, tradeoffs and a solution – they are meant for humans, not tools!

- There are relationships to MDD, however:
  - Patterns can be **implemented** by the generator (after the architect has manually decided to use them!)
  - Patterns & Pattern Languages can help you come up with an architecture and an architectural meta model
    - for a system (Remoting Infrastructures)
    - or a given aspect of a system (Pipes & Filters)

- Always be **sceptical** if you read “this tool supports patterns – all the existing 23“
The Role of Frameworks & Middleware

- **Middleware encapsulates** quite a bit of the accidental system complexity.

- **Example Middleware platforms** include:
  - Various RTOSs such as Osek, QNX, VxWorks
  - CORBA, RT-CORBA, (Lw-)CCM
  - Data Distribution Service (DDS)

- However, even with this level of abstraction, the **remaining complexities are still challenging** enough to make manual work unrealistic.
  - This is especially true for **non-local configuration** where many parameters must be set to compatible values.

- Models and generated code can help work efficiently with powerful and complex platforms, middlewares and frameworks.

Simulation (Systems Execution Modeling)

- **Compose scenarios**
- **Associate performance properties** with scenarios
- **Configure workload generators** to run experiments
- **Feedback results** into models to verify if deployment plan & configurations meet performance requirements
Non-Local Optimizations

• Once systems get really big, it becomes infeasible to manually structure the system according to a set of given constraints.
  • Consider the problem of deploying hundreds of components to a limited set of computing resources so that the overall system latency is minimized and reliability is optimized.

• You can use automatic optimization techniques such as
  • Constraint Solvers based on Predicate Logic (Prolog)
  • Genetic Algorithms

• You can do this with models because the system structure is formally described; the optimizers can work on these models.
  • You might want to use simulation to verify some of the solutions

CONTENTS

• What is MDSD
• What are Embedded Systems
• The Challenges and MDD
  • Resource Efficiency
  • Architectural Consistency
  • System Complexity
  • Product Lines & Variabilities
  • Platform Independence / Portability
  • Tool Integration
  • Hardware & Systems Engineering
• Other Relevant Topics
  • Model-to-Model Transformations
  • Variability Management and Configuration
  • Standards
• Summary
Context

- Embedded systems are often **part of a product line** where products exhibit a **well-defined set of variabilities** among each other, as well as many commonalities.

- These variabilities must be **formally expressed** and need to be **efficiently implemented**.

- **Modeling** can be used to express the variabilities
  - Feature modeling for non-structural variability
  - Custom DSLs for structural variability

- **Code Generation** is a way to implement these variabilities
  - Since no runtime overhead is incurred, this is one of the preferred ways of implementing variabilities in embedded systems

CONTENTS

- What is MDSD
- What are Embedded Systems
- The Challenges and MDD
  - Resource Efficiency
  - Architectural Consistency
  - System Complexity
  - Product Lines & Variabilities
    - **Platform Independence / Portability**
    - Tool Integration
    - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary
Context

- Since embedded systems are not just software, it is often necessary to run the same functionality on different infrastructures:
  - Chips/Architectures
  - Bus Systems
  - Operating Systems
  - OS Configurations: Blocking/Non-Blocking, Scheduler y/n...
  - Threading Libraries
  - Middleware
- ... while the code is (almost) always C ...
- Using #ifdef etc. does not really scale well to large systems and many variabilities...

MDD Solution

- You can program your application logic against a technology-independent API and then generate the "mapping" to the respective hardware.
- In the generated mapping code, you can use all the "tricks" for efficient generated code explained earlier.
- This is a special case of Product Lines that is especially typical – which is why people often don’t consider it PLE.
CONTENTS

• What is MDSD
• What are Embedded Systems
• The Challenges and MDD
  • Resource Efficiency
  • Architectural Consistency
  • System Complexity
  • Product Lines & Variabilities
  • Platform Independence / Portability
  • Tool Integration
    • Hardware & Systems Engineering
• Other Relevant Topics
  • Model-to-Model Transformations
  • Variability Management and Configuration
  • Standards
• Summary

Context

• Application functionality has a long history of being modeled with various formalisms – state machines, data flow, mathematical models
  • With the various tools: Matlab/Simulink, Statemate, ...

• More recently, architectural abstractions (component models, deployment, etc.) are more and more designed using models.

• One challenge is to integrate these different formalisms.

• An example of such an integration is
  • architectural component modeling with UML
  • application logic description with Matlab/Simulink
Solution

- Goal: Transform from a profiled UML2 model into a Matlab/Simulink file
- Transform UML2 to a custom CBD model
- Transform CBD model into Matlab/Simulink model
- Generate .m file that can be imported into Matlab/Simulink to build the actual model
- Various model validation steps in between
- A layouting engine is also integrated (not shown)

Solution II: Why the separate intermediate models?

- Expressive validations can be done at each level
- You don’t have to deal with the intricacies of the UML2 meta model very often – only in the first step!
- You can easily integrate custom editors instead of a UML tool
- It is easy to cascade additional DSL layers (typically more domain specific) on top of the CBD model
Solution III: Cascading DSLs

- Input Models
- MDSD Infrastructure
- Output Models

Domain 1 Model
Functional Domain 1
MDSD Infrastructure

Domain 2 Model
Functional Domain 2
MDSD Infrastructure

Input Models
Basic Technical
MDSD Infrastructure

Code for Target Platform

CONTENTS

- What is MDSD
- What are Embedded Systems
- The Challenges and MDD
  - Resource Efficiency
  - Architectural Consistency
  - System Complexity
  - Product Lines & Variabilities
  - Platform Independence / Portability
  - Tool Integration
  - Hardware & Systems Engineering
- Other Relevant Topics
  - Model-to-Model Transformations
  - Variability Management and Configuration
  - Standards
- Summary
Hardware (Systems Engineering)

- Most embedded systems are not only software – they are a part of a system, i.e. **hardware is also a part** of the game.
  - This is where systems modeling/systems engineering comes in

- Models can represent any concept, not just software, i.e. a **system model can also contain hardware models**
  - hosts, sensors, actuators, networks,
  - each of them described with relevant characteristics such as bandwidth, power, etc.
  - See also: SysML

- You can easily have **software and hardware** (and their relationships) **in the same model**.
  - Optimizations can consider both, if you can describe the hardware characteristics well enough.

„Generating Hardware“

- In contrast to general opinion, it is **possible to generate hardware**.

- Well, sort of.

- The way this works is to:
  - Describe the system to be built using models
  - Generate an “implementation” in e.g. Structured Text
  - “Burn” an FPGA from this specification
    (FPGA = Field Programmable Gate Array)

- This is a **special case of platform independence** – in this case the platform is an FPGA device.
Why M2M

- **M2M transformations are important** to
  - Modularize models
  - Separate concerns in models
  - Keep models on various abstraction levels synchronized
  - Transform models between different tools

- Several different approaches are available: textual/graphical, declarative/procedural/functional

- Tools are becoming **mature**, there are **several** available today:
  - ATL
  - Extend
  - QVT Operations
**Model Merging**

- Several models are **merged** with each other.

- Implications of model merging
  - Typically **easy to implement** (no actual transformation)
  - Meta models are obviously the same
  - Useful if models need to be **modularized** (team issues, performance, ...) and then put together for a complete build

**Model Modifications**

- An existing Model is **modified “in place”**.

- Implications of model modification
  - An existing model is **enhanced at generation time**, adding elements (e.g. the index page of a web page)
  - The model is based on the same metamodel before and after the modification
  - Little initial implementation overhead (e.g. using Java code)
  - Tricky for bigger changes
Model Transformations

• A model is transformed into another model; the input model is left unchanged.

• Implications of model transformations
  • clean separation: separate models, separate metamodels
  • different domains can evolve independently
  • prerequisite for reusing “cartridges”
  • identical copy operations must be programmed explicitly
  • runtime and memory overhead

Model Weaving

• Two (or more) models are woven with each other in a non trivial way (unlike merging)

• Implications of model weaving
  • A weaving model (pointcut) needs to be provided – which is essentially a special kind of transformation rule
  • Homogeneous (shown above) and Heterogeneous Aspects can be woven
  • Most of today’s M2M languages don’t address this use case specifically – weavings must be programmed “imperatively” with the transformation language.
Mixin Models

- The modification or transformation needs to be **parametrized**.

- Implications of mixin models
  - Aka Markup Models
  - Provide **additional (mark up) information** about how a given model should be processed in a modification of transformation
  - Obviously used **together with the other forms**
Variability Analysis

- **Variability analysis** discovers the variable and fixed parts of a product in a domain.
  Parts can be
  - Structural or behavioral
  - Functional or non-functional (technical)
  - Modularized or aspectual

- To define variable parts, we need to have a commonality base: a base platform, a common architecture

- There are two kinds of variability:
  - positive variability: adds something (optional)
  - negative variability: removes something (essential)

- Another classification: **structural** vs. **non-structural** var.

### Structural vs. Non-Structural Variability

- **Structural Variations**
  Example Metamodel

- **Non-Structural Variations**
  Example Feature Models

  Dynamic Size, ElementType: int, Counter, Threadsafe
  Static Size (20), ElementType: String
  Dynamic Size, Speed-Optimized, Bounds Check

- Based on this sample metamodel, you can build a wide variety of models:
Rountine Configuration vs. Creative Construction

- This slide (adopted from K. Czarnecki) is important for the selection of DSLs in the context of MDSD in general:
  - The more you can move your DSL "form" to the configuration side, the simpler it typically gets.
  - We will see why this is especially important for behavior modelling.

Variants and Models II

- It is especially useful to combine structural and non-structural variations
  - Specifically, you may want to "configure" structural models with the help of feature models,
  - We want to describe variants of structural models (and use these variants as generator input)

- Examples:
  - A party may have one or more addresses
  - A party may store telecontacts or not
  - In case of telephone numbers, you may want to store the country code
  - Addresses may have the state field (USA)
You can assign model elements of the structural model to features in the feature model.

The respective model elements are only there if the associated feature is selected,

and it's removed if the feature is not there.

- CONTENTS
  - What is MDSD
  - What are Embedded Systems
  - The Challenges and MDD
    - Resource Efficiency
    - Architectural Consistency
    - System Complexity
    - Product Lines & Variabilities
    - Platform Independence / Portability
    - Tool Integration
    - Hardware & Systems Engineering
  - Other Relevant Topics
    - Model-to-Model Transformations
    - Variability Management and Configuration
    - Standards
  - Summary
(Where) Are standards important?

- **Potential examples** for standardization include Meta meta models, meta models, modeling languages (i.e. MM + Syntax), profiles, tools

- Standards are often **very broad and unspecific to a given domain** (UML, SysML)
  - This might improve interop and tool availability,
  - But also incurs a lot of accidental complexity

- Concrete Advice wrt. **UML**:
  - You might want to use UML for modeling (although specific DSLs are usually better),
  - but you **should not use the UML meta model** for (invisible) intermediate meta models.
  - **Always define your own meta model** independent of UML and then (maybe) map it to a profile!
Summary

- The Embedded community is adopting MDD very quickly (and in some areas it has been done for a long time anyway)

- Many of the advantages of MDD can be exploited especially well in embedded systems

- Tooling is mature and usable... the Eclipse Modeling Project, openArchitectureWare and modern UML tools provide a good starting point.


Some advertisement

- For those, who speak (or rather, read) german:
  Völter, Stahl: Modellgetriebene Softwareentwicklung
  Technik, Engineering, Management
  dPunkt, 2005
  www.mdsd-buch.de

- A very much updated translation is under way:
  Model-Driven Software Development,
  Wiley, Q2 2006
  www.mdsd-book.org