Model-Driven Development
State of the Art

OOP Conference, Monday, January 16th, 2005

Dr Douglas C Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt
Institute for Software Integrated Systems
Vanderbilt University Nashville, Tennessee

Thomas Stahl
t.stahl@bmiag.de
www.bmiag.de
b+m Informatik AG, Chef-Architekt
Kiel, Germany

Markus Völter
voelter@acm.org
www.voelter.de
Independent Consultant for Software Engineering and Technology
Heidenheim, Germany

CONTENTS

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
- An Architectural Process – A Case Study
- Examples of Applying MDD Tools: GME & CoSMIC
- Another Tool: openArchitectureWare
- SOA, Business Process Modeling, & MDD
- Product-line Architecture Case Study
- Summary
Model-Driven Development: State of the Art

CONTENTS

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
- An Architectural Process – A Case Study
- Examples of Applying MDD Tools: GME & CoSMIC
- Another Tool: openArchitectureWare
- SOA, Business Process Modeling, & MDD
- Product-line Architecture Case Study
- Summary

The Road Ahead

CPU & network performance has increased by 3-8 orders of magnitude in past decades

Extrapolating these trends another decade or so yields
- ~100 Gigahertz desktops
- ~100 Gigabits/sec LANs
- ~100 Megabits/sec wireless
- ~10 Terabits/sec Internet backbone

Unfortunately, software quality & productivity hasn’t improved as rapidly or predictably as hardware – especially for distributed real-time & embedded (DRE) systems
Why Hardware Improves So Consistently

Advances in hardware & networks stem largely from R&D on standardized & reusable APIs & protocols

x86 & Power PC chipsets
TCP/IP

Why Software Fails to Improve as Consistently

In general, software has not been as standardized or reusable as hardware

Proprietary & Stovepiped Application & Infrastructure Software

Standard/COTS Hardware & Networks
The Promise

- Develop standardize technologies that:
  1. Model
  2. Analyze
  3. Synthesize &
  4. Provision

complex software systems

The Reality

- Architects (sometimes) use UML to express software designs at a high-level

- Developers write & evolve code manually

We ought/need to be able to do much better than this!
Sources of the Problems

<table>
<thead>
<tr>
<th>Technical Challenges</th>
<th>Inherent &amp; accidental complexities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• More automated specification &amp; synthesis of</td>
</tr>
<tr>
<td></td>
<td>• Broader range of target domain capabilities</td>
</tr>
<tr>
<td></td>
<td>• Model interpreters &amp; transformations</td>
</tr>
<tr>
<td></td>
<td>• Static &amp; dynamic quality of service (QoS) properties</td>
</tr>
<tr>
<td></td>
<td>• Round-trip engineering from models ↔ source</td>
</tr>
<tr>
<td></td>
<td>• Poor support for debugging at the model level</td>
</tr>
<tr>
<td></td>
<td>• Version control of models at the model level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Technical Challenges</th>
<th>Impediments of human nature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Organizational, economic, administrative, political, &amp; psychological barriers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technical Challenges</th>
<th>Ineffective technology transition strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Disconnects between methodologies &amp; production software development realities</td>
</tr>
<tr>
<td></td>
<td>• Lack of incremental, integrated, &amp; triaged transitions</td>
</tr>
</tbody>
</table>

www.cs.wustl.edu/~schmidt/reuse-lessons.html

Key Challenges for Software Developers

Developers & users of software face challenges in multiple dimensions

Logical View

Process View

Use Case View

Physical View

Development View
Key Challenges for Software Developers

**Logical View**

- Determining units of abstraction for system (de)composition, reuse, & validation
- Popular technologies & tools provide inadequate support for:
  - Checking pre-/post-conditions & invariants
  - Specifying & analyzing dependencies
  - Expressing design intent more clearly using domain concepts

**Physical View**

- Integrating/deploying diverse new & reusable application components in a networked environment to ensure end-to-end QoS requirements
- Popular technologies & tools provide inadequate support for:
  - Configuring & customizing components for application requirements & run-time environments
  - Automated mapping of components onto nodes in target environments
Devising execution architectures, concurrency models, & communication styles that ensure multi-dimensional QoS & correctness of new/reusable components

Key Challenges for Software Developers

- Popular technologies & tools provide inadequate support for:
  - Identifying & reducing performance & robustness risks earlier in system lifecycle
  - Satisfying multiple (often conflicting) QoS demands
    - e.g., secure, real-time, reliable
  - Satisfying QoS demands in face of fluctuating/insufficient resources
    - e.g., mobile ad hoc networks (MANETs)

(De)composing systems into reusable modules (e.g., packages, subsystems, libraries) that achieve/preserve QoS properties

- Cyclic dependencies, which make unit testing & reuse hard
- Excessive link-time dependencies, which bloat the size of executables
- Excessive compile-time dependencies, where small changes trigger massive recompiles
Key Challenges for Software Developers

- Popular technologies & tools provide inadequate support for
  - Ensuring semantic consistency & traceability between requirements & software artifacts
  - Visualizing software architectures from multiple views

Promising Solution Approaches

- Multi-faceted Software Development
  - Cross-cutting Concerns
    - Synchronization
    - Persistence
    - Memory Management
    - Fault Tolerance
  - Model-based analysis, generation, & integration & domain-specific languages
  - Verification & validation technologies, e.g., model checking & static analysis

- Component middleware & frameworks that integrate real-time, fault-tolerance, & security properties

- Formalizing best practices & design expertise
Devising composable abstractions whose interfaces & QoS properties can be specified/analyzed via metadata.

- Components encapsulate "business" logic
- Components interact via ports
  - Provided ports, e.g., facets
  - Required ports, e.g., receptacles
  - Event sink & source ports
- Containers provide execution environment
- Components/containers can also communicate via a middleware bus & reuse common middleware services
- Aspect-oriented techniques can help with integration

Promising Solution Approaches

Model-driven development & analysis techniques for optimizing, verifying, & automating the deployment & configuration process.
Promising Solution Approaches

- Synthetic workload & simulated components
- Replaced incrementally with actual applications & components

Software architecture execution modeling/simulation techniques & tools; distributed continuous quality assurance

Process View

- Automate the QA process

Promising Solution Approaches

- Packages view – shows element tree defined by project's build class path
- Type hierarchy view – shows the sub- & supertype hierarchies
- Outline view – shows the structure of a compilation unit or class file
- Browsing perspective – allows navigating models using separate views for projects, packages, types & members
- Wizards for creating elements – e.g., project, package, class, interface
- Editors – syntax coloring, content specific code assist, code resolve, method level edit, import assistance, quick fix & quick assist

Development environments that provide multiple views & minimize dependencies between large-scale software artifacts to optimize development & test cycles
Promising Solution Approaches

- One way to automate tracing between higher-level specifications & lower-level implementations is to leverage model-driven development techniques & tools.

Technology Evolution (1/4)

- Model-Driven Development (MDD)
  - State chart
  - Data & process flow
  - Petri Nets

- Programming Languages & Platforms
  - Operating Systems
  - Hardware
  - C/Fortran
  - Assembly
  - Machine code

- Level of Abstraction
  - Generated Code
  - Platform

- Large Semantic Gap

- Translation
  - Translation
  - Configuration Specification
  - Analysis Tool
Technology Evolution (2/4)

- New languages & platforms have raised abstraction level significantly
  - “Horizontal” platform reuse alleviates the need to redevelop common services

- There are two problems, however:
  - Platform complexity evolved faster than 3rd-generation languages
  - Much application/platform code still (unnecessarily) written manually
    - Particularly for D&C & QA aspects

Technology Evolution (3/4)

- Domain-specific modeling languages
  - ESML
  - PICMCL
  - Mathematic
  - Excel
  - Metamodels

- Domain-independent modeling languages
  - State Charts
  - Interaction Diagrams
  - Activity Diagrams

- OMG is evaluating MDD via MIC PSIG
  - mic.omg.org
Technology Evolution (4/4)

Programming Languages & Platforms
- Needs Automation
- Saturation!!!

Model-Driven Development (MDD)
- Domain-specific modeling languages
  - ESML
  - PICML
  - Mathematic
  - Excel
  - Metamodels
- Research is needed to automate DSLs & model translators

Domain-independent modeling languages
- State Charts
- Interaction Diagrams
- Activity Diagrams

CONTENTS

- Introduction & Motivation
- Definition of Terms
  - Architecture-Centric MDD & Cascading
  - Role of Frameworks & Patterns in the Context of MDD
  - How Generators work – MDD & Compiler Construction
  - Model-To-Model Transformations
  - An Architectural Process – A Case Study
  - Examples of Applying MDD Tools: GME & CoSMIC
  - Another Tool: openArchitectureWare
  - SOA, Business Process Modeling, & MDD
  - Product-line Architecture Case Study
- Summary
The MDD Blueprint

Typically you start with a manually built application

You analyse it with respect to

And code that is completely individual to the application

Code that has the same structure/patterns in all cases

The generic, identical code becomes the platform

Application Models describe the repetitive aspects in an abstract & concise way using a Domain-Specific Language

Transformations create the repetitive code from the application models

Model-Driven Development – State of the Art
27

28
A UML based Ontology for MDD

• Isn’t the OMG/MDA terminology sufficient?
  – Useful as a basis, but perspectives of domain engineering, product lines & software system families missing or weak

• What is an Ontology?
  – Representation of knowledge: definition of concepts & their relationships

• Why UML for that?
  – Provides a popular & standardized notation for concepts (classes) & typical categories of relations (Association, Composition, InstanceOf, Realization, Uses, …)
  – Allows for definition of specific categories through profiling

Domain, Model, & DSL

• A domain describes a bounded area of knowledge or interest
  – It can be structured into various subdomains

• A metamodel is a formal representation of the concepts in that particular (sub-)domain, as well as their relationships
  – A metamodel is also called the abstract syntax (of a DSL)
A Domain-Specific Language (or DSL) comprises:
- The metamodel (the concepts it represents)
- A concrete syntax to represent these concepts
- As well as the semantics of the concepts

A DSL is sometimes called a domain-specific modeling language (DSML)

A formal model (or just "model") built by the DSL:
- is an instance of its metamodel and respects the static semantics
- Uses the concrete syntax of the DSL
- And gets its meaning from the DSLs semantics
A platform supports a domain. Platforms can be cascaded, i.e., they are based on top of each other (see later).

A platform consists of a number of building blocks:
- these can be middleware, libraries, frameworks, components or aspects,
- as well as documentation and tests, of course.

The finished product contains the platform. It consists of generated artifacts, as well as non-generated, i.e., manually implemented artifacts.

Both of these types of artifacts know about & rely on the platform.
• The product – specifically, the generated artifacts – must be built
  – from the formal models describing the system,
  – Using a number of different transformations
• A transformation always uses a formal model as its source, thus it relies on the metamodel on which this formal model is built

Model-to-Model Transformations also produce a formal model as the output
• This is typically based on a different metamodel
• The transformation also relies this target metamodel
• The M2M step can be repeated any number of times in the context of MDD, the models typically becoming more specific to the platform
• Finally, **Model-to-Platform (or model-to-code) transformations** use the formal model to produce the **generated artifacts**

• these rely on & make use of the **platform idioms** ("patterns")

• The Idioms, together with the generated artifacts have to **realize the semantics of the model**
Domain Architecture

- A software system family can be implemented using such a domain architecture.
- In turn, a software system family can be used to support (or realize) product lines.
- A product line then consists of a number of the products built using the software system family’s domain architecture.

MDA Terms

- The Meta Object Facility is the OMG’s Meta Meta Model. XMI is based on top of it, as is UML.
- Action Semantics can be used to add additional semantics & behaviour to models.
- OCL can also add semantics by defining constraints.
MDA Terms

- MDA models are typically (not mandatorily) based on UML + Profiles
- OMG distinguishes between platform specific & platform independent models
- Platform Description models are basically metamodels of the platform

CONTENTS

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
- An Architectural Process – A Case Study
- Examples of Applying MDD Tools: GME & CoSMIC
- Another Tool: openArchitectureWare
- SOA, Business Process Modeling, & MDD
- Product-line Architecture Case Study
- Summary
The domain is **architecturally motivated** e.g., architecture for business software or component infrastructure for embedded systems.

The products to be created are usually complete **applications**, not single components.
The DSL’s metamodel therefore contains architectural concepts that are as abstract as possible i.e., “component” & not “EJB 3 stateless session bean”
The DSL is also called design language. Often, UML profiles are used here, sometimes combined with additional textual specifications.

Typically, the model-to-platform transformation is a template that shows great similarity to the generated code & thus can easily be extracted from a reference implementation.
The transformation does not aim at creating the complete application, but merely an implementation framework containing the architectural infrastructure code (skeleton).

The non-generated, implementation code ("business logic") is manually implemented in the target language (code snippet).

For this purpose, the generated skeleton may contain protected regions for supplementing the application logic that will persist after iterative regeneration.

Alternatively, generated & non-generated code is integrated using suitable design patterns.
Recipe F/W for Integrating Manually Written Code

- Recipe Frameworks help developers to write the “correct” manual code by applying checks on the complete (manual & generated) code.
Recipe F/W for Integrating Manually Written Code

- Recipe Frameworks **help developers to write the “correct” manual code** by applying checks on the complete (manual & generated) code.

The IDE evaluates the checks in the recipe file & points to problems in the manually written code.

Checks that are ok are rendered in **green** (or can be filtered out).

Recipe F/W for Integrating Manually Written Code

- Recipe Frameworks **help developers to write the “correct” manual code** by applying checks on the complete (manual & generated) code.
Recipe F/W for Integrating Manually Written Code

- Recipe Frameworks help developers to write the “correct” manual code by applying checks on the complete (manual & generated) code

You can get more detailed information on the checks executed.

And a nice explaining text that tells developers what to do to fix the problem.
Recipe F/W for Integrating Manually Written Code

- Recipe Frameworks help developers to write the “correct” manual code by applying checks on the complete (manual & generated) code.

You can reevaluate checks at any time.

A “quick fix” button will be added till final release of oAW 4.
Cascading MDD Using Platform Stacking

- The generated code of the lower layer serves as the platform for the next higher level.
- A sequence of generation steps is used, whereas each of the generates code on which the next step builds.

Cascading MDD Using M2M

- Here the higher level models are transformed into lower-level models that serve as input for the lower level generators.
- Model-to-Model Transformations are used.
- Typically, higher level models are more specific to a certain (sub-)domain.
Example for “Business Level” DSL

DYI vs 3rd Party Cartridges

- Do you build your own generator for your specific architecture?
  - This is good, because it's tailored to your architecture
- Or do you want to (re-)use off-the-shelf cartridges for certain standard technologies (such as J2EE, Hibernate, Spring)?
  - You can do the best of both worlds:
    - Define applications using your own metamodels (architecture-centric, maybe functional ones on top)
    - Transform your models to input models for the off-the-shelf cartridges on the lower levels
Overview of Patterns

• Present solutions to common software problems arising within a certain context

• Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs

• Help resolve key software design forces
  - Flexibility
  - Extensibility
  - Dependability
  - Predictability
  - Scalability
  - Efficiency

• Generally codify expert knowledge of design strategies, constraints & “best practices”

MDD tools codify & automate many (but by no means all) aspects of patterns
Overview of Pattern Languages

Motivation
- Individual patterns & pattern catalogs are insufficient
- Software modeling methods & tools largely just illustrate **what/how** – not **why** – systems are designed

Benefits of Pattern Languages
- Define a **vocabulary** for talking about software development problems
- Provide a **process** for the orderly resolution of these problems, eg:
  - What are key problems to be resolved & in what order
  - What alternatives exist for resolving a given problem
  - How should mutual dependencies between the problems be handled
  - How to resolve each individual problem most effectively in its context
- Help to generate & reuse software architectures

Pattern languages are crucial for **domain-specific languages** (DSLs)

Overview of Frameworks

Framework Characteristics
- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications

Application-specific functionality
- Mission Computing
- E-commerce
- Networking
- Database
- GUI
- Scientific Visualization
- Networking

Model-Driven Development – State of the Art
• **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software

• **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts

```java
package org.apache.tomcat.session;
import org.apache.tomcat.core.*;
import org.apache.tomcat.util.StringManager;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
 * Core implementation of a server session
 *
 * @author James Duncan Davidson [duncan@engsun.com]
 * @author James Todd [gonzo@engsun.com]
 */
public class ServerSession {
  private StringManager sm = StringManager.getManager("org.apache.tomcat.session");
  private Hashtable values = new Hashtable();
  private Hashtable appSessions = new Hashtable();
  private String id;
  private long creationTime = System.currentTimeMillis();
  private long thisAccessTime = creationTime;
  private int inactiveInterval = -1;
  ServerSession(String id) {
    this.id = id;
  }
  public String getId() {
    return id;
  }
  public long getCreationTime() {
    return creationTime;
  }
  public ApplicationSession getApplicationSession(Context context, boolean create) {
    ApplicationSession appSession = (ApplicationSession)appSessions.get(context);
    if (appSession == null && create) {
      // XXX
      // sync to ensure valid?
      appSession = new ApplicationSession(id, this, context);
      appSessions.put(context, appSession);
    }
    // XXX
    // make sure that we haven't gone over the end of our
    // inactive interval -- if so, invalidate & create
    // a new appSession
    return appSession;
  }
  void removeApplicationSession(Context context) {
    appSessions.remove(context);
  }
}
```
Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation & deployment of software
- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs & leveraging previous development & optimization efforts
- **Validation reuse**
  - e.g., by amortizing the efforts of validating application- & platform-independent portions of software, thereby enhancing software reliability & scalability

Summary of Pattern, Framework, & MDD Synergies

- **These technologies codify expertise of domain experts & developers**
  - Frameworks codify expertise in the form of reusable algorithms, component implementations, & extensible architectures
  - Patterns codify expertise in the form of reusable architecture design themes & styles, which can be reused event when algorithms, components implementations, or frameworks cannot
  - MDD tools codify expertise by automating key aspects of pattern languages & providing developers with domain-specific modeling languages to access the powerful (& complex) capabilities of frameworks

There are now powerful feedback loops advancing these technologies
Motivation 1

• Why taking a look at compiler construction?

• The core concepts of MDD generators are the same as in compiler construction
  – Formal Languages
  – Transformation of formal artifacts on a higher abstraction level into formal artifacts on lower abstraction level (i.e., raising the abstraction level for programming)
  – Modularity

• Compiler construction is a domain, which is well understood. We should be able to harvest some of the experiences made there: Generators ARE Compilers
Motivation 2

- So compiler construction might give hints for MDD/A tool developers, but why should “users” care about it?
- Construction criteria that have proven good, are selection criteria too.
- We can adopt a clear & unambiguous terminology.
- In the context of MDD we need openness.
  - Adopting or creating own DSLs
  - Adopting or creating own Transformations

That means, the user (i.e., architect) is involved & confronted to some extent with the inner structure of the compile process.

Compiler Construction Blue Print

<table>
<thead>
<tr>
<th>Artifacts</th>
<th>Workflow</th>
<th>Meta-Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Text (e.g., C++ Source-Code)</td>
<td>Compiler-Frontend = Parser</td>
<td>Programming Language</td>
</tr>
<tr>
<td>Abstract Syntax Tree</td>
<td>Checker</td>
<td>Concrete Syntax (e.g., C++ or Java)</td>
</tr>
<tr>
<td></td>
<td>Optimizer</td>
<td>Abstract Syntax</td>
</tr>
<tr>
<td></td>
<td>Compiler-Backend = Codegenerator</td>
<td>Static Semantics (e.g., Declaration of Variables)</td>
</tr>
<tr>
<td>Generated Code</td>
<td></td>
<td>Dynamic Semantics</td>
</tr>
<tr>
<td>Runtime System</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Core Concept: Abstract Syntax

Responsibilities of Abstract Syntax
• Modularity of the compiler: Interface between Frontend & Backend
• Neutralisation of different Concrete Syntaxes (e.g., C and PASCAL)
• Pluggability for different backends (e.g., codegen for Intel/PC or Mainframe)
• Interface for modification of program structure (e.g., optimization)

Mapping of Concepts

<table>
<thead>
<tr>
<th>Compiler Construction (specific concepts)</th>
<th>Model-Driven Software Development (general concepts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Language</td>
<td>A kind of Modeling Language</td>
</tr>
<tr>
<td>Program (Source-Code)</td>
<td>A kind of Model</td>
</tr>
<tr>
<td>Concrete textual Syntax</td>
<td>A kind of DSL</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>Part of Meta-Model</td>
</tr>
<tr>
<td>Static Semantics</td>
<td>Modeling-Constraints</td>
</tr>
<tr>
<td>Program Editor</td>
<td>A kind of DSL Editor</td>
</tr>
<tr>
<td>Parser</td>
<td>Model-Reader</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Model 2 Code Transformation</td>
</tr>
<tr>
<td>Optimization</td>
<td>A kind of Model-2-Model Transformation</td>
</tr>
<tr>
<td>Runtime System</td>
<td>Platform</td>
</tr>
</tbody>
</table>

The plain compiler view on MDD is quite “code-generation centric” & seems to focus textual languages, but in fact MDD takes advantage from generalization & further abstraction of those concepts It's an evolution
So the task of creating a domain architecture from the perspective of compiler construction means:

- Define a formal language with concrete & abstract syntax & static semantics (DSL)
- Implement a Parser for the DSL
- Implement the static semantics
- Implement Modifiers or Transformations based on the abstract syntax (M2M-Transformations)
- Implement a runtime system (Platform)
- Implement one or more backends (M2C-Transformations)

How MDD Tools Can Help

Many of these tasks are supported by generic tools or frameworks, eg:

- Generic DSLs with extension mechanisms (e.g., UML)
- Parser- resp DSL-Editor-Generators
- AS-Modeling & Generation (Metamodel Generators)
- Constraint-Languages (e.g., OCL)
- M2M-Transformation-Languages & Implementations/Interpreters
- M2C-Transformation-Languages & Implementations/Interpreters
- Reusable Meta-Artifacts (Cartridges)
### MDD Tool Blueprint

<table>
<thead>
<tr>
<th>Tool Support (Open Compiler)</th>
<th>Architect’s Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL-Editor Generator</td>
<td>DSL-Model</td>
</tr>
<tr>
<td>Generic DSL-Editor</td>
<td>DSL-Profile</td>
</tr>
<tr>
<td>Parser Generator</td>
<td>DSL-Grammar Spec</td>
</tr>
<tr>
<td>Metamodel Generator</td>
<td>Metamodel</td>
</tr>
<tr>
<td>Constraint Engine</td>
<td>Constraints</td>
</tr>
<tr>
<td>Static Semantics</td>
<td></td>
</tr>
<tr>
<td>M2M Engine</td>
<td>M2M Transformations</td>
</tr>
<tr>
<td>Template Engine</td>
<td>M2C Transformations</td>
</tr>
<tr>
<td>Runtime System</td>
<td>MOOSD-Platform</td>
</tr>
</tbody>
</table>

### Example: openArchitectureWare (oAW)

**oAW – brief introduction**
- Open MDD-Framework (i.e., a framework that enables you to write compilers/generators in the former sense)
- Open Source since 11/2002, contributed to the community by b+m Informatik AG (www.bmiag.de)
- More on that later
Object-orientation is useful for compiler construction, because
- an Abstract Syntax naturally maps to an object-oriented implementation (Meta Classes, Associations …)
- Unmaintainable switch/case statements on syntax element types can be avoided in transformations by using polymorphism and late binding
- For usual compilers (C++, Pascal …), this is a view into the black box. From the viewpoint of the user, the construction details of the compiler are irrelevant
- Note that this does NOT hold in the MDD-context, since the AS/MM is plugged into the framework. So we are in need of an open compiler/generator framework
- Each Meta-Element has the responsibility to translate itself
- Polymorphism can be used at the meta level

Object-Orientation in oAW

Model expressed via Concrete Syntax (e.g., UML-Profile)

MetaModel / Abstract Syntax (oAW: Implementation in Java)

Responsibilities (Object-Oriented)
- Check: Every „EntityObject“ must have at least one „key“-Attribute
- M2C-Template: Generate EJB-Home-Interface
- M2C-Template: Generate Hibernate-Mapping
- Helper/Property: Return a full qualified Java-Classname

Some Responsibilities are implemented in Java (Properties, Checks), others using the oAW-Template language (M2C). They are implemented in or decorated on the respective Meta-Classes
openArchitectureWare – Blueprint

CONTENTS

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
  - An Architectural Process – A Case Study
  - Examples of Applying MDD Tools: GME & CoSMIC
  - Another Tool: openArchitectureWare
  - SOA, Business Process Modeling, & MDD
  - Product-line Architecture Case Study
- Summary
Why You Need M2M

- As explained before, cascading MDD requires model-to-model transformations

```
  ...         ...         ...
  ...         ...         ...
```

- Model for Subdomain 1
- M2M/Code Generator for SD 1
- Programming Model (based on Arch-MM)
- Code Generator for Architectural MDSD Infrastructure
- Code for Target Platform

- Model for Subdomain 2
- M2M/Code Generator for SD 2

Modular, Automated Transformations

- To more easily reuse parts of a transformation, it is a good idea to modularize a transformation
- Note that in contrast to the OMG, we do not recommend looking at, changing or marking the intermediate models
- They are merely a standardized format for exchanging data among transformations
- Example: Multi-Step transformation from a banking-specific DSL to Java via J2EE
Modular, Automated Transformations II

- Example cont’d:

Now consider a Call-Center application; only the first step needs to be adapted

- If both should be transformed to NET, only the backend needs to be exchanged

Transforming “in the Tool”

Developer builds model using for example a UML tool
The XMI produced by the UML tool is parsed by the generator tool – & an AST is created in memory.

Inside the generator, model-to-model transformations are used to build new or modified ASTs. The intermediate ASTs cannot be modified interactively by the developer.
Transforming “in the Tool”

External Model Markings (AO–Modeling)

- To allow the transformation of a source model into a target model (or to generate code) it is sometimes necessary to provide “support” information that is specific to the target meta model
  - Example: Entity Bean vs Type Manager
- Adding these to the source model “pollutes” the source model with concepts specific to the target model
- MDA proposes to add “model markings,” but this currently supported well by very few tools
- Instead, we recommend keeping this information outside of the model (e.g., in an XML file); the transformation engine would use this auxiliary information when executing the transformations
- This is an example of “aspect-oriented programming”
M2M: One Metalevel Higher

- Precondition:
  Representing a class diagram of metalevel n as an object diagram of metalevel n+1

Model-to-Model Transformations: QVT

- Most of the transformations I have built by now have been constructed with **Java code**
  - If the metaclasses have a **well-designed API** (repository API) then this „procedural transformations“ do indeed work well

- However, more & more **dedicated model transformation languages** become available:
  - ATL, MOLA, Wombat (oAW), etc

- The **QVT standard** is becoming a reality it will be finalized by the end of 2006

- QVT actually comprises **three languages**: Relations Language, Black Box Mappings, Core Language
Model–to–Model Transformations: QVT Relational

```plaintext
top relation EntityKeyToTableKey {
   checkonly domain alma entity:Entity {
      key = entityKeyField:Field {};
   };
   enforce domain db table:Table {
      key = tableKey:Key {};
   };
   when {
      EntityToTable(entity, table)
   }
   where {
      KeyRecordToKeyColumns(entity, table)
   }
}
relation PhysicalQuantityTypeToColumn {
   pqName, pqUnit, fieldName : String;
   checkonly domain alma field:Field {
      name = fieldName,
      type = pq:PhysicalQuantityType {
         name = pqName,
         units = pqUnit
      }
   };
   enforce domain db table:Table {
      columns = column:Column {
         name = prefix + fieldName + '_as_' + pqName + '_' + pqUnit,
         type = AlmaPhysicalQuantityTypeToDbType(pq)
      }
   };
   primitive domain prefix: String;
}
```

M2M–Transformations: QVT Operational

```plaintext
mapping DependentPart::part2table(in prefix : String) : Table
inherits fieldColumns {
   var dpTableName := prefix + recordName;
   name := dpTableName;
   columns := mainColumns +
      object Column {
         name := 'key_' + dpTableName;
         type := 'INTEGER';
         inKey := true;
      }
}
end { selfparts->map part2columns(result, dpTableName + '_'); }

query PrimitiveType::convertPrimitiveType() : String =
   if selfname = "int" then 'INTEGER'
   else if selfname = "float" then 'FLOAT'
   else if selfname = "long" then 'BIGINT'
   else 'DOUBLE'
endif endif endif;
```
Transformations in Java: Example Transformation

• With good metaclasses, this works acceptably today

```java
Model createEJBModel( Model source ) {
    Model target = new Model();
    foreach c:Class in sourceclasses {
        ImplementationClass implClass = new ImplementationClass();
        implClass.setName( c.getName()+"Bean" );
        target.addClass( implClass );
        Dependencies define( implClass, c );
        RemoteInterface ri = new RemoteInterface();
        HomeInterface hi = new HomeInterface();
        // set name & add it to target model
        foreach o:Operation in c.Operations {
            // bidirectional, because of generated API
            ri.addOperation( new Operation( o.clone() ) );
            implClass.addOperation( new Operation( o.clone() ) );
        }
    }
    return target;
}
```
Many Means of Transformations

- Today, many means of transformations are used:
  - Plain old Java
  - Eclipse GMT ATL
  - IBM MTF
  - ISIS GReAT
  - Several partial QVT implementations
  - UMLX

- A paper by Czarnecki/Helsen gives a very good overview:
  www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf
Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype
• PHASE 2: Iterate!
• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation
• Summary
Phase 1: Elaborate!

- This first elaboration phase should be handled by a small team, before the architecture is rolled out to the team as a whole.
- We want to build an enterprise system that contains various subsystems such as customer management, billing & catalogs.
- In addition to managing the data using a database, forms & the like, we also have to manage the associated long-running business processes.
- We will look at how we can attack this problem below.
Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype

• PHASE 2: Iterate!

• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation

• Summary

Technology–Independent Architecture

• We decide that our system will be built from **components**
  – Each component can **provide** a number of **interfaces**
  – It can also **use** a number of **interfaces** (provided by other components)
  – Communication is **synchronous**, Communication is also restricted to be **local**
  – We design components to be **stateless**

• In addition to components, we also explicitly support **business processes**
  – These are modeled as a **state machine**
  – Components can trigger the state machine by supplying **events** to them
  – Other components can be triggered by the state machine, resulting in the **invocation of certain operations**
  – Communication to/from processes is **asynchronous**, **remote** communication is supported
Technology–Independent Architecture

• We decide that our system will be built from components
  – Each component can provide a number of interfaces
  – It can also use a number of interfaces (provided by other components)
  – Communication is synchronous, Communication is also restricted to be local

• In addition to components, we also explicitly support business processes
  – These are modeled as a state machine
  – Components can trigger the state machine by supplying events to them
  – Other components can be triggered by the state machine, resulting in the invocation of certain operations
  – Communication to/from processes is asynchronous, remote communication is supported

Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype

• PHASE 2: Iterate!

• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation

• Summary
The programming model uses a **simple Dependency Injection approach** à la Spring to define component dependencies on an interface level.

An **external XML file** takes care of the configuration of the instances.

The following piece of code shows the **implementation of a simple example component**. Note how we use Java 5 annotations.

```java
public @component class ExampleComponent
    implements HelloWorld {
    // provides HelloWorld
    private IConsole console;
    public @resource void setConsole( IConsole c ) {
        this.console = c; // setter for console
    } // component
    public void sayHello( String s ) {
        console.write( s );
    }
}
```

Processes **engines** are components like any other.

For **triggers**, they provide an interface with void operations.

They also define interfaces with the actions that those components can implement that want to be notified of state changes.
Programming Model

• Process Component Implementation Example

```java
public @process class SomeProcess
    implements ISomeProcessTrigger {
    private IHelloWorld resource;
    public @resource void setResource( IHelloWorld w ) {
        this.resource = w;
    }
    public @trigger void T1( int procID ) {
        SomeProcessInstance i = loadProcess( procID );
        if ( guardG1() ) {
            // advance to another state...
        }
    }
    public @trigger void T2( int procID ) {
        SomeProcessInstance i = loadProcess( procID );
        // ...
        resourcesayHello( "hello" );
    }
}
```

Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – **Technology Mapping**
  – Mock Platform
  – Vertical Prototype
• PHASE 2: Iterate!
• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation
• Summary
Technology Mapping

- For the remote communication between business processes we will use web services
  - From the interfaces such as IHelloWorld, we generate a WSDL file, & the necessary endpoint implementation We use on of the many available web service frameworks
- **Spring will be used** as long as no advanced load balancing & transaction policies are required

```xml
<beans>
  <bean id="proc" class="somePackageSomeProcess">
    <property name="resource"><ref bean="hello"/></property>
  </bean>
  <bean id="hello" class="somePackageExampleComponent">
    <property name="console"><ref bean="cons"/></property>
  </bean>
  <bean id="cons" class="someframeworkStdOutConsole">
  </bean>
</beans>
```

- Once this becomes necessary, we will use **Stateless Session EJBs**
  The necessary code to wrap our components inside beans is easy to write

Technology Mapping

- **Persistence** for the process instances – like any other persistent data – is managed using **Hibernate**
  - To make this possible, we create a **data class for each process**
  - Since this is a normal value object, using Hibernate to make it persistent is straight forward

*Decide about standards usage here, not earlier*

*But keep in mind: First solve the problem Then look for a standard Not vice versa*
Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype

• PHASE 2: Iterate!

• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation

• Summary

Mock Platform

• Since we are already using a PROGRAMMING MODEL that resembles Spring, we use the Spring container to run the application components locally

• Stubbing out parts is easy based on Springs XML configuration file

• Since persistence is something that Hibernate takes care of for us, the MOCK PLATFORM simply ignores the persistence aspect
Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation
- Summary

Vertical Prototype

- The **vertical prototype** includes parts of the customer & billing systems
  - For creating an invoice, the billing system uses **normal interfaces** to query the customer subsystem for customer details
  - The invoicing process is based on a **long-running process**
- A **scalability test** was executed & resulted in two problems:
  - For short running processes, the repeated loading & saving of persistent process state had become a problem. **A caching layer** was added
  - Second, web-service based communication with process components was a problem. **Communication was changed to CORBA** for remote cases that were inside the company
Vertical Prototype

• The vertical prototype includes parts of the customer & billing systems
  – For creating an invoice, the billing system uses normal interfaces to query the customer subsystem for customer details
  – The invoicing process is based on a long-running process
• A scalability test was executed & resulted in two problems:
  – For short running processes, the repeated loading & saving of persistent process state had become a problem. A caching layer was added
  – Second, web-service based communication with process components was a problem. Communication was changed to CORBA for remote cases that were inside the company

WORK ON PERFORMANCE IMPROVEMENTS HERE, NOT EARLIER

It is bad practice to optimize design for performance from the beginning, since this often destroys good architectural practice.

In certain domains, there are patterns to realize certain QoS properties (such as stateless design for large-scale business systems). You shouldn’t ignore these intentionally at the beginning.

Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype
• PHASE 2: Iterate!
• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation
• Summary
Phase 2: Iterate!

- Spring was intended for the production environment

- **New requirements** (versioning!) have made this infeasible
  - Spring does not support two important features: **Dynamic installation/de-installation** of components,
  - & **isolations of components** from each other (classloaders)

- Eclipse has been chosen as the new execution framework The PROGRAMMING MODEL did not change; the TECHNOLOGY MAPPING, however, had to be adapted

Architectural Case Study

- **PHASE 1: Elaborate!**
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype

- **PHASE 2: Iterate!**

- **PHASE 3: Automate!**
  - **Architecture Metamodel**
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation

- Summary
Components can provide & require interfaces

Interfaces have operations – they're defined as usual
Architecture Metamodel

A new version of an interface has to have the same return type and the same parameters - or parameters with subtypes.

If a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A.

A process component’s process is described using a state machine.
Architecture Metamodel

If a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A.

A container runs a number of components.

A new version of an interface has to have the same return type and the same parameters - or parameters with subtypes.

And the triggers are special kinds of operations.

If a component B is a new version of a component A, then B has to have the same interfaces, additional provided interfaces, fewer required interfaces or new version of interfaces of A.

A new version of an interface has to have the same return type and the same parameters - or parameters with subtypes.
Model-Driven Development – State of the Art

Architecture Metamodel

Constraints are used to define the semantics of versioning

Architectural Case Study

- PHASE 1: Elaborate!
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- PHASE 2: Iterate!
- PHASE 3: Automate!
  - Architecture Metamodel
  - Glue Code Generation
  - DSL-based Programming Model
  - Model-based Architecture Validation
- Summary
Glue Code Generation

- Our scenario has several useful locations for glue code generation
  - We generate the **Hibernate** mapping files
  - We generate the **web service & CORBA adapters** based on the interfaces & data types that are used for communication The generator uses reflection to obtain the necessary type information
  - Finally, we generate the **process interfaces** from the state machine implementations

- In the programming model, we use **Java 5 annotations** to mark up those aspects that cannot be derived by using reflection alone
- Annotations can help a code generator to “know what to generate” without making the programming model overly ugly

Architectural Case Study

- **PHASE 1: Elaborate!**
  - Technology-Independent Architecture
  - Programming Model
  - Technology Mapping
  - Mock Platform
  - Vertical Prototype
- **PHASE 2: Iterate!**
- **PHASE 3: Automate!**
  - Architecture Metamodel
  - Glue Code Generation
  - **DSL-based Programming Model**
  - Model-based Architecture Validation
- **Summary**
DSL–based Programming Model

- We use DSLs for components, interfaces & dependencies
  Describing this aspect in a model has two benefits:
  - First, the GLUE CODE GENERATION can use a more semantically rich model as its input,
  - & the model allows for very powerful MODEL-BASED ARCHITECTURE VALIDATION (see below)

- From these diagrams,
  - we can generate a skeleton component class
  - all the necessary interfaces
- Developers simply inherit from the generated skeleton & implement the operations defined by the provided interfaces
DSL–based Programming Model

Type Model

```
<component> AddressManager
  <interface> AddressStore
    add( p: Person) : void
    addAddress( p: Person, a: Address) : void
    getAddresses( p: Person) : Address[]
```

Composition Model

```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

System Model

```
<systems>
  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  </system>
  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  </system>
</systems>
```

Type model defines components (which are instantiatable types), interfaces & data types, as well as the dependencies among them.

Composition Model defines named configurations of component instances & the wiring among them.
Model-Driven Development – State of the Art

**DSL–based Programming Model**

- **Type Model**
  - `<component>` AddressManager
  - `<entity>` Person
    - `name: String`
    - `firstName: String`

- **Composition Model**
  - `<configuration>` addressStuff
    - `<deployment>` am
      - `<wire>` personDAO
  - `<configuration>` customerStuff
    - `<deployment>` cm
      - `<wire>` addressStore

- **System Model**
  - `<system>` production
    - `<node>` server
  - `<node>` client

- **System model deploys configurations onto systems & nodes Nodes define the kind of system they represent**

- **Using Cascaded MDD, we generate**
  - **DAO Components** for Entities from the Entities in the model
  - An **interface** for the DAO component,
  - As well as the **implementation code** for the DAO & the Entity itself

- **SomeEntity**
  - `<transform>` SomeEntityDAO
  - `<generate>` SomeEntity-DAO.java

- **SomeEntityDAO**
  - `<generate>` SomeEntity-DAOBase.java
  - `<generate>` SomeEntity-DAO.java

Model-Driven Development – State of the Art
• We also use cascading for the Process Components

They also model a trigger interface for that component with no operations

First, developers model the process component itself
Model-Driven Development – State of the Art

DSL–based Programming Model

• We also use **cascading** for the Process Components

Using M2M, the operations are **derived from the triggers** used in the state machine.

Developers then model the **state machine** for that process component & associate it with the process component.

Using M2M, the operations are derived from the triggers used in the state machine.

Using M2M, the operations are derived from the triggers used in the state machine.

Model-Driven Development – State of the Art

141

142
• We also use **cascading** for the Process Components

Using M2M, the Entity that stores process instances persistently is derived from the state machine; then the Entity transformations kick in – see before.

As usual, from components we generate **skeleton base classes**.
Instead of letting developers implement the business logic manually, we generate an “Intermediate” class that contains the executable, & persistence-aware state machine.

Finally, developers extend that intermediate class & implement guard and action operations manually by overriding abstract methods.
Architectural Case Study

• PHASE 1: Elaborate!
  – Technology-Independent Architecture
  – Programming Model
  – Technology Mapping
  – Mock Platform
  – Vertical Prototype

• PHASE 2: Iterate!

• PHASE 3: Automate!
  – Architecture Metamodel
  – Glue Code Generation
  – DSL-based Programming Model
  – Model-based Architecture Validation

• Summary

Model-Based Architecture Validation

• It is checked that
  – for triggers in processes there is a component that calls the trigger
  – Dependency management: It is easy to detect circular dependencies among components
  – Components are assigned to layers (app, service, base) & dependencies are only allowed in certain directions

• The component signature generated from the model prevents developers from creating dependencies to components that are not described in the model
Another really important aspect in our example system is **evolution of interfaces**:

```
<<component>>
SomeCompV1

<<interface>>
SomeInterface
soSomething(int, ValueObject)

<<component>>
SomeCompV2

<<interface>>
AnotherInterface

<<component>>
SomeCompV3

<<interface>>
SomeInterfaceV3
soSomething(int, ValueObjectV2)
anAdditionalOperation()

<<vo>>
ValueObject

<<vo>>
ValueObjectV2
```

**Model–Based Architecture Validation**

**CONTENTS**

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
- An Architectural Process – A Case Study

**Examples of Applying MDD Tools:**

- GME & CoSMIC
- Another Tool: openArchitectureWare
- SOA, Business Process Modeling, & MDD
- Product-line Architecture Case Study
- Summary
Model-Driven Development – State of the Art

Generic Modeling Environment (GME)

```
"Write Code That Writes Code That Writes Code!"
```

GME Architecture

**Application Developers (Modelers)**
- Browser
- COM
- GModel
- GMeta
- XML
- ODBC

**MDD Tool Developers (Metamodelers)**
- Add-On(s)
- Translator(s)
- COM
- GME Editor
- Metamodel
- Paradigm Definition
- Storage Options

Supports “correct-by-construction” of software systems

GME is open-source: www.isis.vanderbilt.edu/Projects/gme/default.htm

---

MDD Application Development with GME

• **Application developers** use modeling environments created w/MetaGME to build applications
  – Capture elements & dependencies visually

Model-Driven Development – State of the Art 152
**MDD Application Development with GME**

- **Application developers** use modeling environments created with MetaGME to build applications
  - Capture elements & dependencies visually
  - Model interpreter produces something useful from the models
    - e.g., 3rd generation code, simulations, deployment descriptions & configurations

**MDD Tool Development in GME**

- **Tool developers** use MetaGME to develop a domain-specific graphical modeling environment
  - Define syntax & visualization of the environment via metamodeling
• **Tool developers** use MetaGME to develop a domain-specific graphical modeling environment
  
  – Define syntax & visualization of the environment via metamodeling
  
  – Define static semantics via **Object Constraint Language (OCL)**
  
  – Dynamic semantics implemented via model interpreters
**Applying GME to System Execution Modeling**

System Execution Modeling Workflow

1. Compose scenarios to exercise critical system paths/layers
2. Associate performance properties with scenarios & assign properties to components specific to paths/layers
3. Configure workload generators to run experiments, generate path-layer-specific deployment plans, & measure performance along critical paths/layers
4. Feedback results into models to verify if deployment plan & configurations meet performance requirements

**Motivation: Service-Oriented Architectures**

- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols
Applications

Service-Oriented Architecture Middleware

Operating System & Communication Protocols

Hardware Devices

Motivation: Service–Oriented Architectures

- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols
- Traditional methods of development have been replaced by middleware layers to reuse architectures & code
  - Viewed externally as *Service-Oriented Architecture (SOA) Middleware*

Multilayer Resource Manager (MLRM)

Operating System & Communication Protocols

Hardware Devices

Motivation: Service–Oriented Architectures

- Historically, distributed real-time & embedded (DRE) systems were built directly atop OS & protocols
- Traditional methods of development have been replaced by middleware layers to reuse architectures & code
  - Viewed externally as *Service-Oriented Architecture (SOA) Middleware*
- e.g., DARPA Adaptive & Reflective Management System (ARMS) program’s *Multi-layer Resource Manager (MLRM)*
  - MLRM leverages standards-based SOA middleware to manage computing & communication resources for shipboard computing environments
ARMS Multi–Layer Resource Manager (MLRM)

- Domain-specific layered architecture includes:
  - Top domain layer contains components that interact with the mission manager of TSCE
  - Middle resource pool layer is an abstraction for a set of computer nodes managed by a pool manager
  - Bottom resource layer contains the actual resource computing components

Serialized Phasing is Common in Large-scale Systems

System infrastructure components developed first

Application components developed after infrastructure is mature
Serialized Phasing is Common in Large-scale Systems

Integration Surprises!!!

Complexities of Serialized Phasing

- System infrastructure cannot be tested adequately until applications are done
Complexities of Serialized Phasing

Complexities
- System infrastructure cannot be tested adequately until applications are done
- Entire system must be deployed & configured properly to meet QoS requirements
- Existing evaluation tools do not support “what if” evaluation

Unresolved QoS Concerns with Serialized Phasing

Key QoS concerns
- Which D&C’s meet the QoS requirements?
Unresolved QoS Concerns with Serialized Phasing

Key QoS concerns
- Which D&C’s meet the QoS requirements?
- What is the worse/average/best time for various workloads?
- How much workload can the system handle until its QoS requirements are compromised?

It is hard to address these concerns in processes that use serialized phasing.
Promising Solution Approach: New Generation of System Execution Modeling (SEM) Tools

Validate Design Rules
• System will adhere to system design specifications
  • “Correct-by-construction”

Validate Design Conformance
• System will be properly deployed & configured to resemble system design rules

Conduct “What If” Analysis
• QoS concerns can be analyzed prior to completing the entire system
  • e.g., before system integration phase

The cycle is repeated when developing application & infrastructure components

Our Approach: Emulate Application Behavior via QoS-enabled SOA Middleware & MDD Tools

Component Workload Emulator (CoWorker)
Utilization Test Suite Workflow (CUTS):
While creating target infrastructure
1. Use a domain-specific language (DSL) to define & validate infrastructure specifications & requirements
2. Use DSL to define & validate application specifications & requirements
3. Use middleware & MDD tools generate D&C metadata so system conforms to its specifications & requirements
4. Use analysis tools to evaluate & verify QoS performance
5. Redefine system D&C & repeat

Enable testing on target infrastructure early in development lifecycle
Motivation for Using Emulation

- Can use actual target infrastructure
  - Rather than imprecise simulations
- Many artifacts can be used directly in the final production system
  - e.g., models of application component D&C plans
- Early feedback to developers, architects & systems engineers
  - Instead of waiting for to complete application components before conducting performance experiments

Our SOA Middleware & MDD Tool Infrastructure

- **System Design & Specification Tools**
  - Define & validate system specification & requirements
- **System Assembly & Packaging Tools**
  - Compose implementation & configuration information into deployable assemblies
- **System Deployment Tools**
  - Automates the deployment of system components & assemblies to component servers
- **Component Implementation Framework**
  - Automates the implementation of many system component features

PICML & CIAO & DAnCE

CUTS

www.dre.vanderbilt.edu/CIAO & www.dre.vanderbilt.edu/cosmic
D&C & Performance Requirements
- Critical path deadline is 350 ms
- Main sensor to main effector through configuration
- Components in the critical paths must be deployed across all 3 hosts
- Main sensor & effector must be deployed on separate hosts

Questions we would like to answer
1. Can we meet the D&C & performance requirements?
Questions we would like to answer

1. Can we meet the D&C & performance requirements?
2. Are there multiple deployments that meet the 350ms critical path deadline?
   • Which yields the most headroom
3. Can we meet the 350ms critical path deadline with all component deployed on a single host?
Summary of CUTS Challenges

PICML Model of SLICE Scenario

1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems
3. Non-intrusive benchmarking & evaluation
4. Simplifying component behavior specification

Emulate its behavior?

Single-point of data collection?

Define behavior non-programmatically?
Summary of CUTS Challenges

PICML Model of SLICE Scenario

1. Evaluate QoS characteristics of DRE systems
2. Emulate QoS characteristics of DRE systems
3. Non-intrusive benchmarking & evaluation
4. Simplifying component behavior specification
5. Simplify component customization
6. Precise analysis of performance

Customizing generic components?

End-to-end execution

Time critical
Challenge 1: Evaluating QoS Characteristics of Enterprise DRE Systems Early in Life-cycle

**Context**
- In phase 2 of ARMS, MLRM is implemented using Real-time CCM (via CIAO & DAnCE)
- In phase 1 of ARMS, QoS evaluation was not done until integration
  - Prolonged project development
- Software components & challenges are similar in both phases

**Problem**
- Evaluate MLRM QoS earlier in development process
  - e.g., prior to integration
Solution: Evaluate Component QoS & Behavior using Component–based Emulators

- System components are represented as Component Workload Emulators (CoWorkErs)
- Each CoWorkEr is a CCM assembly component constructed from CCM monolithic components
- Each CoWorkEr has an optional database
  - Can be local or remote
- CoWorkErs can be interconnected to form operational strings

---

Challenge 2: Emulating Behavior & QoS of Enterprise DRE Systems

**Context**
- In phase 1 of ARMS, QoS evaluation was not done until integration
- QoS test was done using ad hoc techniques
  - e.g., creating non-reusable artifacts & tests that do not fully exercise the infrastructure

**Problem**
- Emulating behavior & QoS in a reusable manner so we can easily exercise the complete infrastructure, & apply to other contexts
Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs

Emulate workloads, e.g., CPU, database & memory

Perform background workloads
Solution: Emulate Component Behavior & QoS Using Configurable CoWorkErs

Receive events from CoWorkErs

Send events to CoWorkErs
Challenge 3: Non-Intrusive Benchmarking & Evaluation

Context
- The SLICE scenario of MLRM is composed of multiple components deployed over multiple nodes
- Each component, including components in assemblies, must be evaluated

Problem
- Collecting data from each component without interfering emulation
- Collecting data within skewing the performance metrics

Solution: Decouple Emulation & Benchmarking

- CUTS environment is decoupled into two sections
  - Emulation & benchmarking

Emulation

Benchmarking
Solution: Decouple Emulation & Benchmarking

- CUTS environment is decoupled into two sections
  - Emulation & benchmarking
- Data acquisition is done in 2 phases & at lower priorities than emulation
  1. **BenchmarkAgent** collects performance metrics
  2. **BenchmarkAgent** submits data to the **BenchmarkDataCollector** at user-defined intervals

- **Each CoWorkEr has a BenchmarkAgent**
Challenge 4: Simplify Characterization of Workload

Context
• Persons implementing the SLICE scenario come from different disciplines
  – e.g., software architects, developers, & systems engineers
• CUTS users may not be familiar with 3rd generation or configuration languages
  – e.g., C++ & Java or XML, respectively

Problem
• Providing alternative methods to programming CoWorkEr behavior & creating dense configuration files

Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

- Workload Modeling Language (WML) is used to define the behavior of CoWorkEr components
Solution: Use Domain–Specific Modeling Language to Program CoWorkEr Behavior

- Workload Modeling Language (WML) is used to define the behavior of CoWorkEr components.
- WML events represent different types of workloads in CoWorkEr.

Actions are attached to events & specified in order of execution to define workload strings:
- Each action has attributes, e.g., number of repetitions, amount of memory to allocate & etc.

Attributes for CPUAction:

Startup workload

Event-driven workload

Workload string
Solution: Use Domain-Specific Modeling Language to Program CoWorkEr Behavior

- Workload Modeling Language (WML) is used to define the behavior of CoWorkEr components.
- WML events represent different types of workloads in CoWorkEr.
- Actions are attached to events and specified in order of execution to define workload strings.
    - Each action has attributes, e.g., number of repetitions, amount of memory to allocate & etc.
- WML is interpreted to XML characterization files.
- Characterization specified in CoWorkEr.

Challenge 5: Simplify Component Customization

Context
- Default CoWorkEr can send & receive every type of event.
- The SLICE components are each different & all do not send/receive the same types of events.
    - Each contains a different composition.

Problem
- How can we customize CoWorkEr components without requiring modification & recompilation of components?
Solution: Customize CoWorkErs at System Modeling Level

- Event sinks of a CoWorkErs are delegated to the respective event sources of the EventHandler
- Events produced by the EventProducer are delegated to respective events sources for a CoWorkErs

Default CoWorkErs

Solution: Customize CoWorkErs at System Modeling Level

- Event sinks of a CoWorkErs are delegated to the respective event sources of the EventHandler
- Events produced by the EventProducer are delegated to respective events sources for a CoWorkErs
- Delegated event sources & sinks can be removed from CoWorkErs
  - Does not require recompilation of components

Custom CoWorkErs

Event sources removed

Event sinks removed
Challenge 6: Precise Analysis of QoS Performance

Context
- There are many components in SLICE & combinations in the D&C configuration these components

Problem
- How can we assist users in pinpointing problematic areas in
  - D&C?
  - End-to-end QoS of mission-critical paths?

- Too much workload?
- Too many components?

Missed deadline
Solution: Present Metrics Graphically in Layers to Support General & Detailed Analysis

- BenchmarkManagerWeb-interface (BMW) analyzes & graphically displays performance metrics

- General analysis of actions

- BMW General Time Data

- e.g., transmission delay & processing
Solution: Present Metrics Graphically in Layers to Support General & Detailed Analysis

- **BenchmarkManagerWeb-interface (BMW)** analyzes & graphically displays performance metrics
- General analysis shows users overall performance of each CoWorkEr
  - e.g., transmission delay & processing
- Detailed analysis shows users the performance of an action in the respective CoWorkEr
  - e.g., memory & CPU actions, event handling & etc

Green means end-to-end deadline met
Applying CUTS to the SLICE Scenario

Using ISISLab as our target infrastructure

1. Use PICML to define & validate infrastructure specifications & requirements
2. Use WML to define & validate application specifications & requirements
3. Use DAnCE to deploy component emulators on target infrastructure
4. Use CUTS to evaluate & verify QoS performance
5. Redefine system D&C & repeat

Defining Components of SLICE Scenario in PICML for CUTS

- Each component in SLICE is defined as a CoWorkEr
- The default CoWorkEr is customized to handle events specific to its representative SLICE component
- Each CoWorkEr is assigned a unique user-defined ID number
- The benchmark data submission rate is set to 15 seconds
Defining Behavior of SLICE Scenario Components using WML

<table>
<thead>
<tr>
<th>Effector 1 &amp; Effector 2</th>
<th>Workload performed every second</th>
<th>CPU: 25 reps</th>
<th>PUBLISH: STATUS – SIZE 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload performed after receipt of command event</td>
<td>CPU: 25 reps</td>
<td>PUBLISH: STATUS - SIZE 256</td>
<td></td>
</tr>
</tbody>
</table>

Overall Results of SLICE Scenario

Test 11 produced the best results
- Average case: 221 ms
- Worse case: 343 ms

Population size of 11 tests

- Only 4 of 11 deployments met the 350 ms critical path deadline for average time
- Test 11 only test to meet critical path deadline for worse time
SLICE Scenario Results: Meeting D&C & QoS Requirements

<table>
<thead>
<tr>
<th>Test</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>sensor 1 &amp; planner 1</td>
<td>planner 2, configuration,</td>
<td>Sensor 2, error recovery &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp; effector 1</td>
<td>effector 2</td>
</tr>
<tr>
<td>10</td>
<td>sensor 1 &amp; planner 1</td>
<td>configuration &amp; effector 1</td>
<td>planner 2, sensor 2, error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>recovery &amp; effector 2</td>
</tr>
<tr>
<td>11</td>
<td>sensor 1, planner 1 &amp;</td>
<td>planner 2 &amp; effector 1</td>
<td>sensor 2, error recovery &amp;</td>
</tr>
<tr>
<td></td>
<td>configuration</td>
<td></td>
<td>effector 2</td>
</tr>
</tbody>
</table>

- Test 9 meet the critical path QoS requirement, but did not meet the deployment requirement
- Tests 10 & 11 meet both the critical path QoS & deployment requirements

We were able to answer the critical path & deployment questions

SLICE Scenario Results: Single Host Deployment

<table>
<thead>
<tr>
<th>Test</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>All components</td>
<td>(nothing)</td>
<td>(nothing)</td>
</tr>
<tr>
<td>5</td>
<td>All components, excluding error recovery</td>
<td>Error recovery</td>
<td>(nothing)</td>
</tr>
<tr>
<td>6</td>
<td>All component in the critical path</td>
<td>All components not in the critical path</td>
<td>(nothing)</td>
</tr>
</tbody>
</table>

Test 6 had a average time of 323 ms

We were able to answer the question about deploying on a single node
Summary of Lessons Learned

- SOA middleware technologies allowed leveraging the behavior & functionality of target architecture for more "realistic" emulations
- SOA technologies allowed us to focus on the "business" logic of CoWorkErs
  - D&C handled by the underlying MDD & middleware technology

- CUTS allowed us to test deployments before full system integration testing
- CUTS allowed us to rapidly test deployments, which would have take longer using ad hoc techniques
  - e.g., hand-coding the deployment & configuration of components

Level of Abstraction

Development Timeline
Summary

- We motivated the need for the Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS).
- We presented an DRE system example that used CUTS to evaluate deployments before complete integration.
- We presented the design & implementation of CUTS, along with the R&D challenges we faced.

www.cs.wustl.edu/~schmidt/PDF/CUTS.pdf
www.cs.wustl.edu/~schmidt/PDF/QoSPML-WML.pdf
### Modeling Frontends

- Frontends (parsers) are pluggable
- **a large number of UML modeling tools**
  (Rose, MagicDraw, Enterprise Architect, Innovator, Poseidon, XDE, …)
- MS Visio integration
- Simple parsing of XML files
- **Integration with parser generators**
  (such as antlr, JavaCC)
- **GEMS integration**
  (UVanderbilt/ISIS’s Generic Eclipse Modeling System)

### Industry Standard Support

- **Natively provides a Java-based Metametamodel**
  - with a UML-based metaclass generator that can optionally create Hibernate-persistent metaclasses;
  - constraints implemented in Java based on a nice library

- **Can work with EMF-based models & metamodels**
  - provides workflow adapters for EMF-based transformation tools (such as ATL) or editors (such as GMF)

- **Can use Netbeans MDR as a metadata repository**
Workflow Control

- **ant-based plugin framework**

- **Workflow Engine**
  - controls when which processing step is executed
  - parameterizable through property files

- **Cartridges**
  modularized processing steps ("macros") including resources such as templates, transformations, metaclasses

```xml
<workflow>

<!-- Include property file -->
<property file="workflowproperties"/>

<!-- Component for parsing XMI files -->
<component id="xmiParser" class="orgopemfXmiReader">
  <modelFile value="${modelFile}"/>
  <metaModelPackage value="dataDataPackage"/>
  <outputSlot value="model"/>
</component>

<!-- Component for cleaning directories -->
<component id="dirCleaner" class="orgopemmonDirectoryCleaner">
  <directories value="${srcGenPath}"/>
</component>

<!-- Component for generating code -->
<component id="generator" class="orgoped2Generator">
  <metaModel class="orgopenarchiemfEmfMetaModel">
    <metaModelPackage value="dataDataPackage"/>
  </metaModel>
  <expression value="modelget(0)"/>
  <startDefine value="templates::Root::Root"/>
  <outputStrategy class="orgopenaroutputBeautifyingOutput">
    <genPath value="${srcGenPath}/"/>
    <srcPath value="${srcGenPath}/"/>
  </outputStrategy>
</component>

</workflow>
```

Cartridges

- **A cartridge is a self-sustained “piece of tool”**
  i.e., it comes with templates, transformations, workflow snippets, metaclasses

- **Real, Deep modularization**
  a cartridge can extend other cartridges by
  - extending metaclasses
  - overriding templates
  - advising templates (before, after, around)
  - providing new methods for existing metaclasses
Code Generation

- powerful & elegant template language (XPand)
  - template modularization
  - template polymorphism
  - template overriding
  - AOP support for template programming
  - powerful expression language (OCL-Like)

- Advanced Eclipse Editor
  (syntax coloring, find-references, code completion, static error checks)

Recipe Framework

- Checks manually written code for compliance with a defined programming model
  - Based on the model, checks are instantiated
  - The IDE (Eclipse) verifies the checks & provides quick fixes
Model-to-Model Transformation

- **model transformation** as well as model-enrichment/completion

- **Java code can be used for transformations**
  very rich APIs & libraries are provided to make this easy

- **WOMBAT:**
  
  Textual, functional language
  powerful collections support, same expression language as in XPand

  Can operate on all supported metamodels
  (EMF, MDR, oAW classic) & transform between them

- **Integration with Industry-Standard M2M tools** such as
  ATL for EMF metamodels

---

Status / Track Record / Future

- **Version 3.1 is current**

- **Proven track record in various domains & project contexts**
  telcos, internet, enterprise, embedded realtime, finance, …

- **Version 4 will be released end of Jan, 2006**
  - Parts are already in use by some developers
  - First milestone build Dec 15, 2005

- **www.openarchitectureware.org**

  **LIVE DEMO!**
What is SOA?

- The whole concept of SOA is **not really well defined** today. There are a couple of ideas:
  - SOA (at least in practice) means **web services**
  - SOA means that everything is run through **one middleware infrastructure** instead of many
  - SOA means, that IT is more **business** department-oriented
- Rather, SOA is an **architectural style** to build **scalable** (performance as well as size), **maintainable & manageable** systems
SOA vs. Component – Based Architecture

- A well-done component-based architecture with well-defined interfaces & clearly cut component responsibilities can quite justifiable be considered SOA
  - Components are a natural choice as the building blocks that provide & consume services, specifically, since component platforms already separate functionality from infrastructure!
  - The industry currently defines a standard for Service-Component Architectures [SCA]
Characteristics of SOAs

- Service interactions are typically (though not mandatorily) message-oriented, or document centric
  - Instead of defining rigidly typed interfaces, document structures (schemas) are defined that serve as the basis for interactions
  - This can make evolution of message structures & versioning much simpler

- The interaction patterns, i.e., valid sequences of messages are explicitly defined
  - Interactions are often conversational, i.e., conversational session state is kept “at both sides” of a service interaction
  - This allows orchestration among services
  - Usually, interactions are asynchronous

Characteristics of SOAs II

- Quality of service aspects are explicitly addressed
  - Service providers do not just provide a certain services’ functionality, they provide the functionality with a define service level (performance, reliability, etc)

- Service descriptions & characteristics are available at runtime
  - Using service registries systems can be assembled dynamically

- Often, services are interoperable - they can be used by systems implemented on various platforms

- Services should be designed to be coarse grained & encapsulate functionality relevant from “business perspective” (although nobody can say what this really means)

- Services are typically (but by no means exclusively) used by explicitly modeled business processes

- They are secure, transactional & manageable (like any good system)
• MDD is ideally suited to describe & develop these kinds of systems

The signatures & data structures used in the service can be using models

The valid interactions can be defined using models (e.g., protocol state machines)
Relationship to MDD

- MDD is ideally suited to describe & develop these kinds of systems.

Quality of Service Contracts can be defined using models – code to watchdog them can be generated.

Component-based Development, as well as the assignment of service providers & service consumers can be based on models.
• MDD is ideally suited to describe & develop these kinds of systems

The Mappings of the Service Messages/Calls to communication infrastructures (web services, Tibco, JMS, ) can be automatically generated

Glue code for the deployment to various execution platforms can be generated automatically
**Relationship to MDD**

- MDD is ideally suited to **describe & develop** these kinds of systems

The runtime repository can be **populated from the data in the models**

**Metamodel for SOA/CBD Development**

- This metamodel describes
  - component types & interfaces
  - operations & data structures (not shown)
  - component dependencies
Metamodel for SOA/CBD Development

• This one describes component instances & their "wiring"

<table>
<thead>
<tr>
<th>Component name</th>
<th>Interface name</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>target</td>
</tr>
<tr>
<td>required Interface name</td>
<td>wiring</td>
</tr>
</tbody>
</table>

Configuration name

Component Instance name

Wire name

context Wire inv

for each type of component instance requirement, there must be a wire of the same name.

Metamodel for SOA/CBD Development

• This final viewpoint associates component instances with nodes & containers for deployment & glue code generation

<table>
<thead>
<tr>
<th>Configuration name</th>
<th>Component Instance name</th>
<th>Wire name</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Composition Stuff

System Stuff

Node name

Container name

context ComponentInstance inv

the type of the target instance must provide the interface pointed to by the wire's source's target.
Metamodel for SOA/CBD Development

- Dependencies between viewpoint models are only allowed in the way shown below in order to
  - Be able to have several compositions per type model
  - And several system models per composition

- This is important to be able to have several “systems”,
  - Several deployed locally for testing, using only a subset of the defined components,
  - And “the real system”

A Simple Metamodel for SOA – with CBD

- Here we define the notion of a service more extensively

- Note how services are connected to the components defined in the previous set of metamodels

- Note that the SCA standard uses the same metaphors: components, instances, deployment, services
Business Process Modeling & SOA

- BPM comes **top down**, SOA goes **bottom up** (driven by WS-technology)
- There is a **non-empty intersection**: Declaration of BPs, infrastructure software (BPE, ESB). Approaches are not harmonized yet (e.g. BPEL vs. BPML/N)
- However, **there are significant synergies** – especially when using separation of concerns. BPM needs SOA but not the other way round

The business processes are typically **modeled**, as is their relationship to the service (interfaces) they use
Business Process Modeling & SOA

• BPM comes top down, SOA goes bottom up (driven by WS-technology)

• There is a non-empty intersection: Declaration of BPs, infrastructure software (BPE, ESB). Approaches are not harmonized yet (e.g. BPEL vs. BPML/N)

• However, there are significant synergies – especially when using separation of concerns. BPM needs SOA but not the other way round

The important thing is that you are able to map the same services to different middleware infrastructures – & use all of them from the business processes Again a reason to use models!

Note that it is often unrealistic to have one & only one middleware because of varying non-functional requirements
Model-Driven Development: State of the Art

CONTENTS

- Introduction & Motivation
- Definition of Terms
- Architecture-Centric MDD & Cascading
- Role of Frameworks & Patterns in the Context of MDD
- How Generators work – MDD & Compiler Construction
- Model-To-Model Transformations
- An Architectural Process – A Case Study
- Examples of Applying MDD Tools: GME & CoSMIC
- Another Tool: openArchitectureWare
- SOA, Business Process Modeling, & MDD

Product-line Architecture Case Study

Summary

Case Study Example: Boeing Bold Stroke

- Avionics mission computing product-line architecture for Boeing aircraft
- DRE system with 100+ developers, 3,000+ software components, 3-5 million lines of C++

- Based on COTS hardware, networks, operating systems, languages, & middleware
Applying COTS to Boeing Bold Stroke

COTS & standards-based middleware, language, OS, network, & hardware platforms
- Real-time CORBA middleware services
- ADAPTIVE Communication Environment (ACE)
- C++/C & Real-time Java
- VxWorks operating system
- VME, 1553, & Link16
- PowerPC

www.cs.wustl.edu/~schmidt/TAO.html

Benefits of Using COTS
- Save a considerable amount of time/effort compared with handcrafting capabilities
- Leverage industry “best practices” & patterns in pre-packaged & ideally standardized form
Limitations of Using COTS

- QoS of COTS components is not always suitable for mission-critical systems
- COTS technologies address some, but not all, of the domain-specific challenges associated with developing mission-critical DRE systems

What we need is a reuse technology for organizing & automating key roles & responsibilities in an application domain

Motivation for Product-line Architectures (PLAs)

Legacy DRE systems have historically been:
- Stovepiped
- Proprietary
- Brittle & non-adaptive
- Expensive
- Vulnerable

Consequence:
- Small HW/SW changes have big (negative) impact on DRE system
- QoS & maintenance
**Motivation for Product-line Architectures (PLAs)**

- **Frameworks** factors out many reusable general-purpose & domain-specific services from traditional DRE application responsibility.
- Essential for **product-line architectures (PLAs)**
- Product-lines & frameworks offer many configuration opportunities
  - e.g., component distribution & deployment, user interfaces & operating systems, algorithms & data structures, etc.

**Overview of Product-line Architectures (PLAs)**

- PLA characteristics are captured via **Scope, Commonalities, & Variabilities (SCV) analysis**
  - This process can be applied to identify commonalities & variabilities in a domain to guide development of a PLA
- Applying SCV to Bold Stroke
  - **Scope** defines the domain & context of the PLA
    - Bold Stroke component architecture, object-oriented application frameworks, & associated components, e.g., GPS, Airframe, & Display
Applying SCV to the Bold Stroke PLA

- **Commonalities** describe the attributes that are common across all members of the PLA family
  - Common object-oriented frameworks & set of component types
    - e.g., GPS, Airframe, Navigation, & Display components
  - Common middleware infrastructure
    - e.g., Real-time CORBA & a variant of Lightweight CORBA Component Model (CCM) called Prism

- **Variabilities** describe the attributes unique to the different members of the family
  - Product-dependent component implementations (GPS/INS)
  - Product-dependent component connections
  - Product-dependent component assemblies (e.g., different weapons systems for different customers/countries)
  - Different hardware, OS, & network/bus configurations

**Patterns & frameworks** are essential for developing reusable PLAs
Applying Patterns & Frameworks to Bold Stroke

Reusable object-oriented application domain-specific middleware framework
- Configurable to variable infrastructure configurations
- Supports systematic reuse of mission computing functionality
- 3-5 million lines of C++
- Based on many architecture & design patterns

Patterns & frameworks are also used throughout COTS software infrastructure

Legacy Avionics Architectures

Key System Characteristics
- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 usecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

Avionics Mission Computing Functions
- Weapons targeting systems (WTS)
- Airframe & navigation (Nav)
- Sensor control (GPS, IFF, FLIR)
- Heads-up display (HUD)
- Auto-pilot (AP)

1: Sensors generate data
2: I/O via interrupts
3: Sensor proxies process data & pass to missions functions
4: Mission functions perform avionics operations
### Legacy Avionics Architectures

**Key System Characteristics**
- Hard & soft real-time deadlines
  - ~20-40 Hz
- Low latency & jitter between boards
  - ~100 usecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

**Limitations with Legacy Avionics Architectures**
- Stovepiped
- Proprietary
- Expensive
- Vulnerable
- Tightly coupled
- Hard to schedule
- Brittle & non-adaptive

### Decoupling Avionics Components

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O driven DRE application</td>
<td>• Tightly coupled components</td>
<td>• Apply the Publisher-Subscriber architectural pattern to distribute periodic, I/O-driven data from a single point of source to a collection of consumers</td>
</tr>
<tr>
<td>• Complex dependencies</td>
<td>• Hard to schedule</td>
<td></td>
</tr>
<tr>
<td>• Real-time constraints</td>
<td>• Expensive to evolve</td>
<td></td>
</tr>
</tbody>
</table>

#### Structure

- **Publisher**
  - produce
- **Event Channel**
  - attachPublisher
  - detachPublisher
  - attachSubscriber
  - detachSubscriber
  - pushEvent
- **Subscriber**
  - consume
  - filterEvent

#### Dynamics

- **Publisher**
  - produce
  - pushEvent
- **Event Channel**
  - attachSubscriber
  - detachSubscriber
- **Subscriber**
  - consume
Applying the Publisher-Subscriber Pattern to Bold Stroke

Bold Stroke uses the Publisher-Subscriber pattern to decouple sensor processing from mission computing operations:

- Anonymous publisher & subscriber relationships
- Group communication
- Asynchrony

Considerations for implementing the Publisher-Subscriber pattern for mission computing applications include:

- **Event notification model**
  - Push control vs pull data interactions
- **Scheduling & synchronization strategies**
  - e.g., priority-based dispatching & preemption
- **Event dependency management**
  - e.g., filtering & correlation mechanisms

---

Ensuring Platform-neutral Inter-process Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
</table>
| Mission computing requires remote IPC | Applications need capabilities to:  
  - Support remote communication  
  - Provide location transparency  
  - Handle faults  
  - Manage end-to-end QoS  
  - Encapsulate low-level system details | **Apply the Broker architectural pattern** to provide platform-neutral comms between mission computing boards |
| Stringent DRE requirements | | |

---

**Context**

```
Client Proxy
  - marshal
  - unmarshal
  - receive_result
  - service_p
  - calls
  - exchange
  - call_service_p
  - start_task

Structure

Broker
  - main_loop
  - srv_registration
  - srv_lookup
  - xml_message
  - manage_QoS
  - exchange
  - calls

Server Proxy
  - marshal
  - unmarshal
  - dispatch
  - receive_request
```

---

Model-Driven Development – State of the Art
Ensuring Platform-neutral Inter-process Communication

<table>
<thead>
<tr>
<th>Context</th>
<th>Problems</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mission computing</td>
<td>• Applications need capabilities to:</td>
<td>• Apply the Broker architectural pattern to provide platform-neutral</td>
</tr>
<tr>
<td>requires remote IPC</td>
<td>• Support remote communication</td>
<td>comms between mission computing boards</td>
</tr>
<tr>
<td>Stringent DRE</td>
<td>• Provide location transparency</td>
<td></td>
</tr>
<tr>
<td>requirements</td>
<td>• Handle faults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Manage end-to-end QoS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Encapsulate low-level system details</td>
<td></td>
</tr>
</tbody>
</table>

Dynamics

Applying the Broker Pattern to Bold Stroke

Bold Stroke uses the Broker pattern to shield distributed applications from environment heterogeneity, e.g.,

- Programming languages
- Operating systems
- Networking protocols
- Hardware

A key consideration for implementing the Broker pattern for mission computing applications is QoS support

- e.g., latency, jitter, priority preservation, dependability, security, etc
Benefits of Patterns

- Enables reuse of software architectures & designs
- Improves development team communication
- Conveys “best practices” intuitively
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

Limitations of Patterns

- Require significant tedious & error-prone human effort to handcraft pattern implementations
- Can be deceptively simple
- Leaves many important details unresolved

We therefore need more than just patterns to achieve systematic reuse
Applying Frameworks to Bold Stroke

Framework benefits & characteristics

- Frameworks exhibit “inversion of control” at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are “semi-complete” applications

www.cs.wustl.edu/~schmidt/Ace.html

Limitations of Frameworks

- Frameworks are powerful, but can be hard to develop & use effectively
- Significant time required to evaluate applicability & quality of a framework for a particular domain
- Debugging is tricky due to inversion of control
- V&V is tricky due to “late binding”
- May incur performance degradations due to extra (unnecessary) levels of indirection

We therefore need something simpler than frameworks to achieve systematic reuse

www.cs.wustl.edu/~schmidt/PDF/Queue-04.pdf
**Product-line component model**

- Configurable for product-specific functionality & execution environment
- Single component development policies
- Standard component packaging mechanisms
- 3,000+ software components

**Benefits of Component Middleware**

- Creates a standard “virtual boundary” around application component implementations that interact only via well-defined interfaces
- Define standard container mechanisms needed to execute components in generic component servers
- Specify the infrastructure needed to configure & deploy components throughout a distributed system
Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware

Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use
Limitations of Component Middleware

- Limit to how much application functionality can be refactored into reusable COTS component middleware
- Middleware itself has become hard to provision/use
- Large # of components can be tedious & error-prone to configure & deploy without proper integration tool support

There are many middleware technologies to choose from
Model-driven development (MDD)

- Apply MDD tools to
  - Model
  - Analyze
  - Synthesize
  - Provision middleware & application components
- Configure product-specific component assembly & deployment environments
- Model-based component integration policies

www.isis.vanderbilt.edu/projects/mobies

www.rl.af.mil/tech/programs/MoBIES/
Benefits of MDD

- Increase expressivity
  - e.g., linguistic support to better capture design intent
- Increase precision
  - e.g., mathematical tools for cross-domain modeling, synchronizing models, change propagation across models, modeling security & other QoS aspects
- Achieve reuse of domain semantics
  - Generate code that’s more “platform-independent” (or not)!
  - Support product-line architecture development & evolution

Limitations of MDD

- Modeling technologies are still maturing & evolving
  - i.e., non-standard tools
- Magic (& magicians) are still necessary for success
Open MDD R&D Issues

- **Accidental Complexities**
  - Round-trip engineering from models ↔ source
  - Mismatched abstraction levels for development vs debugging
  - Tool chain vs monolithic tools
  - Backward compatibility of modeling tools
  - Standard metamodeling languages & tools

- **Inherent Complexities**
  - Capturing specificity of target domain
  - Automated specification & synthesis of
    - Model interpreters
    - Model transformations
    - Broader range of application capabilities
    - Static & dynamic QoS properties
  - Migration & version control of models
  - Scaling & performance
  - Verification of the DSLs

* Solutions require validation on large-scale, real-world systems*
Positive Development

- **Today’s tools can be used productively** – although sometimes some “magic” is necessary
  - Today’s problem is not really that we need better tools, we rather need more experience!
- **Standardization efforts** are slowly coming to fruition: EMF, QVT, MIC

  **Start today – it will make you more productive**

- **CoSMIC** is available from www.dre.vanderbilt.edu/cosmic
- **GME** is available from www.isis.vanderbilt.edu/Projects/gme/default.htm
- **openArchitectureWare** is available from www.openarchitectureware.org

Some Advertisements

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

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