Executable UML for Model Driven Architecture
Executable UML update

- Raising the level of abstraction – Some history & benefits.

- Short about Executable UML and Model Driven Architecture (MDA)

- PIM vs PSM Separation and PIM Adaptation.
  (separation of concerns)

- Testing and translation of Executable UML models.
  (= model compilers)

- Information about some performance benchmarking and results.

- Short information about some other projects and learnings.
Raising the level of abstraction
Raising the level of abstraction

PIM’s becomes reusable large-scale components, possible to reuse across different platforms.

Our Model Compilers becomes reusable large-scale assets for our platforms. (our competitive advantage)

Executable UML models are runnable and testable without generating any code. (~like PowerPoint)
Platform Independent Models (PIM) translated to target code by Model Compilers for different platforms.

The Shlaer-Mellor Method for OOA evolves to Executable UML. (with an Action Language for UML)
Model Driven Architecture (MDA) introduced. Defines the importance of PIM vs. PSM separation.

Increased reuse of larger target code libraries. (example: Enterprise Java Beans)

UML Model Based Development = drawing pictures & write target language code in them.
“Generate” complete code with behavior from models or write all the code by hand.
Models get tied to the underlying execution environment (platform), hard to reuse.

Increase of Model Based Development, followed by Java, Ada95 and C++.
(drawing some pictures and then write all code by hand)

SW programming tooling & compiler improvements. (more, quicker, better)

Higher-level languages and OO SW methods.
Some more efficient SW reuse enabled.
Writing & maintaining software gets more efficient.
(moving away from CPU specific assembly)

Assembly code dominates the embedded arena.
(hieroglyphs "carved in stone" = no real reuse)
Higher levels of abstraction - Benefits

- Executable UML and MDA allows us to:
  - Express more functionality with less effort.
  - Focus on modeling functionality (not code).
  - Test more with less effort.
  - Reuse more with less effort.
  - Reduce the number of faults. (ref. snapshot TR’s)

- Fly higher, better, faster and safer:
  - Higher degree of automation.
  - Higher degree of reuse.
  - Higher quality. (ref. Coverity for SW Quality & Security Analysis)
Higher degree of automation example

Storage of data in a Non-Volatile Storage
(important data that must survive a system restart, power failure etc.)

Sample Plex-C code in AXE

All required by the Plex-C designer is to mark the variable as `reload`. The Plex-C compiler and the AXE platform handles the rest. In AXE all data marked as `reload` is reloaded on system restart.

```plaintext
record InstrMeasCalcResultFile;
    string variable IMCR_Mnemonic 7 ds reload static;
    variable          IMCR_GMCA1 R ds;
    variable          IMCR_GMCA2 R ds;
    variable          IMCR_GMCA2 R ds;
    variable          IMCR_GMCA2 R ds;
    variable          IMCR_GMCA2 R ds;
    variable          IMCR_IsVisible 16 ds reload static;
    variable          IMCR_AddWithOther 16 ds reload static;
    variable          IMCR_AddWithOther 16 ds reload static;
    variable          IMCR_AddIndex 16 ds;
    variable          IMCR_Occurrences R ds;
    variable          IMCR_Percentage R ds;
    variable          IMCR_SortIndex 16 ds;
    variable          NOO_TotOccur (2) R ds;
end record;
```

Sample model in Executable UML

All required by the model designer is to mark the variable as `reload`. The Model Compiler handles the rest. All data marked as `reload` is reloaded from the file system on system restart. Verified on CPP node Q2-09.
Executable UML and Model Driven Architecture (MDA)
What is Executable UML?

- Executable UML is a graphical specification language, combining a well-defined subset of UML with executable action semantics and rules for timing.
- An evolutionary stage of the Shlaer-Mellor Method.
- Executable UML Specifications are platform independent, can be run, tested and debugged much like a program but long before any code is generated.
- Executable UML models are translated into design by application independent Model Compilers.
- Executable UML = executable models without generating code (= execute models right out of the repository).
- In Mentors xtUML: $x = \text{Executable}$ and $t = \text{Translatable}$
Executable UML Specifications

Execution on all levels.

Radio Access Network (RAN)

RNC, RBS, MediaGateWay, HLR, Cellular Phone etc.

The system and its parts

The "old" Domain Chart

Plug-In of Interface Compatible xtUML Models (leaf-components)

DOMAIN LEVEL

NETWORK LEVEL

NODE LEVEL

SYSTEM LEVEL

SYSTEM PART LEVEL

DOMAIN LEVEL
UML Forum - Robot Contest in Tokyo

Robot contest in full swing

The globetrotter

Reviewing robot contest descriptions

Leon Starr - www.modelint.com
xtUML Translation according to MDA

**Model Driven Architecture**
Platform Independent Models (PIM) are translated into design through Platform Specific Models (PSM).

**Platform Independent Model**
The application models are free from implementation (design) details and fully reusable across different existing and future platforms and for different SW/FW/HW design alternatives.

**Platform Specific Model**
Populated metamodel of the software architecture. (the "PSM metamodel")

The translator becomes the expert system for how to generate the most efficient code for the target platform. Embeds target language and platform expertise, best design practices etc.

The above has been conceptually proven in Ericsson projects.
Example PIM by Steve Mellor
Translation to different platforms

Translating the same xtUML model using different Model Compilers

Erlang

Plex-C for APZ 21230/-33

C/C++ for Windows, OSE & Linux

Java
Translation to different platforms

VHDL for simulation in ModelSim
PIM vs PSM Separation

(separation of concerns)
PIM vs PSM Separation

- If the purpose of the PIM is the ability to implement it on multiple target platforms and execution environments, to achieve a cost efficient reuse across different platforms, it is of utmost importance to keep the PIM free from PSM concepts. (= separation of concerns)

- That way we separate the definition of the solution to the problem (PIM) from the definition of its implementation (PSM), handled as separate subject matters.

- What we then get is a **Generic Component** that can run on different platforms.

- The PIM provides clean (PIM’ish) interfaces with clear protocols to the outside world and makes no assumptions on execution environment, target platform or other lower level design details, like programming language stuff.

- The PSM on the other hand knows about CPU cores, shared memory, Operating System, middleware resources, target programming language etc.
PIM vs PSM Separation

This post-it note was used in most PIM vs PSM discussions.
PIM vs PSM Separation

- We define our PIM component with clear interfaces & protocols. (defined sequences)
- We build static structures and define behaviour with State Machines & Action Language.
- The developers focus only on modeling the functionality. (not on modelling the code)
- The weather inside the PIM is nice and sunny – It’s a happy world, free from PSM stuff.

We also model a test environment and run our test cases by executing the PIM itself to verify its behavior.

We now have a PIM that we need to integrate with legacy interfaces in a legacy environment, which basically always is the case.

We can do this integration in different ways.
1) Adding PSM concepts to the PIM

- We can start adding PSM concepts to the model to capture the PSM view of the problem.
- We may also add programming language code, make Operating System calls etc.
- We tie the PIM closer to the underlying target platform, thus short circuiting the PSM.
- Our PIM is wrecked and can not be reused on different platforms in a cost efficient way.

Each time the PSM concepts are updated we need to create a new version of our model.

We may end up with different models for the same thing but for different platforms, or for different versions of a platform.

This basically means no real, efficient reuse.

(mostly rework)
2) Preserving the PIM vs PSM Separation

- Build an adaptation layer outside the PIM for handling the interface adaptations.
- This will keep the PIM free from PSM pollution and stay reusable across different platforms.
- On the PIM side we don’t care how the adaptations are made, only that they are! (but not by the PIM)
- The adaptation layer itself is platform specific and not designed for reuse on other platforms.

When adding new adaptations we do not need to update the PIM.

When testing new adaptations we do not need to retest the PIM.

Keeping adaptations separated from the PIM means that cross-platform reuse of the PIM is preserved.

Adaptations need to be updated only when interfaces are updated.
The Adaptation Layer is designed separate from the PIM, with tailor made parts for each interface. Some interface adaptations are very simple, while other are more complex and will require some more complex solutions, or combinations of solutions. (new marks, model compiler updates, glue code etc.) The complexity of each adaptation mainly depends on how good or bad the interfaces match.

On a perfect interface match, the adaptation is very thin. Only a simple wiring straight to the legacy interfaces. More complex adaptations may require combined solutions, and also include updates of the model compiler.

Modelled adaptation. Probably has PSM related concepts in it, but can be modelled, run and tested just like a PIM.

Hand written glue code is simple and straight forward, but may require some complex logic.
PIM Adaption Layer

- If adaptations are modelled in Executable UML *, they can be tested on host by running them. (just like you run PowerPoint slides)

- If all adaptations are modelled, the Executable UML * PIM can be tested on host by running it, with or without the Adaption Layer.

- A modelled Adaption Layer containing PSM related concepts in it is really a Platform Adapted Model. (PAM)

- Any modelled and platform-adapted-PIM with PSM stuff in it is really a PAM, and such models are not really designed for reuse across different platforms.

- A model one may think is a PIM, but which has been polluted with PSM related constructs and dependencies e.g. for splitting some processing on a multi-core platform (or any other PSM-constructs for that matter) is really a PAM.

- Such PSM-style constructs should better be handled by an Adaption Layer designed for the multi-core platform. But it can be modelled! 😊

Executable UML
Model Testing
and
Model Translation
Model Based Testing of Components

A real example of Model Based Testing for the Diameter Base Protocol component.

A modelled testbench is connected to DBP.

The testbench contains modelled test cases used for driving the test while testing the DBP.

While running the test we can get printouts of results, error reports, inspect intermediate results, set breakpoints etc. like in a traditional source code debugger.

Chained test cases can be used to build a fully automated regression test suite.
xtUML Modeling & Translation

xtUML Model Translation
The /// Model Compiler

Host or Target Platform
Platform Specific Code

PSC

Compiled Executable

Runtime coverage information for PIM Model Based Test Coverage
(supported by the Model Compiler)

Marking Model
Translation Rules
Design Patterns

xtUML Repository
xtUML Runtime (VM)

class
attribute
datatype
state
transition
event

message
instance
value
queue
stack

Model compilation flags for tuning of performance etc.

Modeled Test env. for Model Based Regression Tests

xtUML Modeling & Testing
BridgePoint UML Suite

xtUML Modeling & Testing
BridgePoint UML Suite

PIM

Application Model

PIM

xtUML Repository
xtUML Runtime (VM)

EAB/FJP Toni.Siljamaki@ericsson.com

Executable UML, BridgePoint, xtUML and MDA
2011-11-28
Wiring model to legacy example

To integrate generated code with legacy code we need marking support for doing so in the xtUML Model Compiler.

Here are some examples of marking support developed for our proprietary Model Compiler for wiring generated code straight to existing code in our products.

Wiring for integration with legacy is supported for:

- communication ports
- datatypes
- classes
- signals

...and is also supported for:
- functions
- class operations
- port messages
- datatype packages
- and more...

The marks (= model compiler flags) for integration with legacy are applied to the model prior to translating the model into code. The xtUML Model Compiler then takes care of this wiring for seamless integration with existing legacy code.
Performance Benchmarking
Performance Benchmarking

- Performance is key!

- MDD is of limited use if it doesn't meet the performance requirements (budget) for real-time critical applications.

- Low performance of the generated code drives the manufacturing costs for HW.

- This is the main reason to why performance benchmarking is one of the first things our Design Units do.
AAL2 Performance Comparison: Hand-coded vs. BridgePoint xtUML (AAL2 Setup & Release)

Benchmarking against 10 year old existing code

CPU Load %

xtUML 1st attempt: 9% 72 us/conn.
xtUML 1st opt.: 7.5% 59 us/conn.
xtUML 2nd opt.: 6.3% 51 us/conn.
Original AAL2 NCC: 6% 48 us/conn.

Code is generated by Ericsson Proprietary xtUML Model Compiler developed @ PDU CPP
Benchmarking newly developed SW

The Radical SW Pilot @ Ericsson PDU CPP

- A new SW function was developed in multiple tracks:
  - Hand-written C++ (synchronous design)
  - Modelled using Executable UML (xtUML with Action Language)
  - Modelled using UML with C++ (not ready yet)

- The C++ was functionally tested both on host and target.

- The Executable UML model was developed in BridgePoint xtUML and was functionally tested on host using a modelled test environment of *Scenario Players* controlled by a *Conductor*.

- 100% of the code was generated from the xtUML model.
  (some minor glue code was written by hand)
Modelled UDPSH Component
Modelled UDPSH Test Environment
Some UDP SH Class & State Diagr.
UDPSH BridgePoint xtUML version: Model Based Test Coverage

Tested using Model Based Regression Test
(modelled test scenario players)

The else-branch coverage indicates the amount of alternative execution paths tested in the Action Language code.
# UDPSH Traffic Load Test Cases

<table>
<thead>
<tr>
<th>Test #</th>
<th>Pre-condition 1</th>
<th>Pre-condition 2</th>
<th>Load Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>P2a: sess in new gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>P2b: sess in exist gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>P2a: sess in new gr</td>
<td>TC2: 2500 sess/sec</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>P2b: sess in exist gr</td>
<td>TC2: 2500 sess/sec</td>
</tr>
<tr>
<td>5</td>
<td>P1a: 1500s in 50gr</td>
<td>P2a: sess in new gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>6</td>
<td>P1a: 1500s in 50gr</td>
<td>P2b: sess in exist gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>7</td>
<td>P1b: 1500s in 1500gr</td>
<td>P2a: sess in new gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>8</td>
<td>P1b: 1500s in 1500gr</td>
<td>P2b: sess in exist gr</td>
<td>TC1: 1500 sess/sec</td>
</tr>
<tr>
<td>9</td>
<td>P1c: 3000s in 3000gr</td>
<td>P2a: sess in new gr</td>
<td>TC3: 600 sess/sec</td>
</tr>
<tr>
<td>10</td>
<td>P1c: 3000s in 3000gr</td>
<td>P2b: sess in exist gr</td>
<td>TC3: 600 sess/sec</td>
</tr>
</tbody>
</table>

**Static load in memory**

**Runtime load**
UDPSH Traffic Load Test Case

5 clients setting up sessions in 10 different groups with 50 sessions each = 2500 sessions per sec.
UDPSH Performance Benchmarking

Test Case 3

5 clients setting up sessions in new groups, load is 2500 sessions/sec
Worst-case CPU load for these 10 test cases

Worst-case CPU load for these 10 test cases

Code is generated by Ericsson Proprietary xtUML Model Compiler
UDPSH Performance Benchmarking

Test Case 9

5 clients setting up sessions in **new groups**, load is 600 sessions/sec

Static pre-condition load is 3000 sessions in 3000 groups

**Worst-case memory cost for these 10 test cases**

![Graph showing memory utilization](image)

- **Hand-coded C++**: 901 KB
- **BridgePoint UML+AL**: 908 KB
- **UML with C++**: 7 KB (not ready yet)

Code is generated by Ericsson Proprietary xtUML Model Compiler
HOT POTATO test: Sending data packets between state machines

Full tests were performed on different HW platforms & OS’es.
(PowerPC vs. Intel and OSE vs. Linux SW/HW platforms)
HOT POTATO signalling benchmark: BridgePoint vs. a UMLwithC++ tool

Signalling between modelled state machines using 100 Byte packages.
(platform is 8572 with Monta Vista Linux)
HOT POTATO signalling benchmark: BridgePoint vs. a UMLwithC++ tool

Signalling between modelled state machines using 100 Byte packages.
(platform is “monster machine” 5570/Sles/gcc64)

Round-trip time in us

- BridgePoint single thread: 0.11 us
- BridgePoint dual thread: 0.36 us
- UMLwithC++ tool single thread: 0.47 us
- UMLwithC++ tool dual thread: 4.5 us

Dual core vs. single core
BridgePoint runtime call graph

Ericsson proprietary xtUML Model Compiler tailor made for small footprint & highest performance.
UMLwithC++ tool runtime call graph

Off-the-shelf UML C++ code generator with adapted runtime system.
Short information about some other projects
Diameter Base Protocol Project

- Multi-site project
- Focus on PIM reuse
- Run DBP on different platforms
- Model Compiler for new platform
SS7 SCTP Base Protocol Project

- Phase 1: Benchmark characteristics.
- Phase 2: Run SCTP on a different platform.
AAL5 Modelling Project

- Develop AAL5 for new platform
- Newly trained MDD team
- "Compete" with hand-coding team

- Characteristics benchmarking
- Lead-time benchmarking
- Quality benchmarking
Summer Worker Project

- 17 years young guy
- 1w reading & training
- 3 half-day workshops
- 3w modelling
- 2w PowerPoint pres.
Why MDD? - The MDD Vision

HAND-CODING VS. MODEL DRIVEN DESIGN

- It’s easy to hide a bad design in hand-written code.
- When building models you are forced to expose your analysis to others.

THE MDD VISION

- Ensure **productivity** by raising to higher abstraction levels. (= models)
  - Focus on modeling functionality rather than code.
  - Less documents and handovers as “the model is the source”.
  - Models are executable specifications to be translated into highly efficient code with small footprint and low costs in CPU load.
- Excellent **quality** in development.
  - First time right.
- Cost-efficient **reuse** of software.
- Support for **agile** development.
What have we learned from xtUML and MDA?

- "Surprisingly" high quality of software.
  - Typically no faults on target after model execution & testing on host.
  - Coverity did not find any problems in generated code.

- Generated code equal in performance with hand-optimized code.
  - Results from benchmarking of CPU load and memory costs.

- Smaller refactoring of models is easier than expected.
  - Example: Move control from one state machine to another, or to new ones.
  - Need a modelled test bench for retesting the refactored model.

- Full control of the entire code generation process.
  - Generated code easily integrated with legacy products.
  - Flexibility to alter code generation whenever needed.
  - Automated support for Trace & Error in generated code.
  - Not dependent on tool vendor.

- Increasing the level of abstraction not necessarily means getting a reduction in performance. (= old myth killed)

- It’s time to stop just talking about MDD, because...
“It takes a whole new way of thinking to solve the problems that were created by the old way of thinking”

Albert Einstein