Executable UML for Model Driven Architecture
Executable UML update

- Raising the level of abstraction – Some history & benefits.
- Executable UML and Model Driven Architecture (MDA)
- PIM vs PSM Separation and PIM Adaptation. (separation of concerns)
- Open translation of Executable UML models. (= model compilers)
- More of everything!!! ...with less effort = lower costs (”extra allt” 😊 )
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)

2000's

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)

1990's

Assembly code dominates the embedded arena.
(hieroglyphs "carved in stone" = no real reuse)

1980's

1970’s
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)

But there has always been a 10-15 years slack in "real" acceptance & deployment of new technology. Why?
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

```pascal
// 2.4.4.1 Record Object InstrMeasCalcResult
record InstrMeasCalcResultFile;
  string variable IMCR_Mnemonic 7 ds reload static;
  variable       IMCR_GMCA1 R ds;
  variable       IMCR_GMCA2 R ds;
  variable       IMCR_GMCB1 R ds;
  variable       IMCR_GMCB2 R ds;
  variable       IMCR_IsVisible 16 ds reload static;
  variable       IMCR_AddWithOther 16 ds reload static;
  variable       IMCR_AddIndex 16 ds reload static;
  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 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.

- 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

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)

Execution on all levels.

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
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

- 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.

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.

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

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

Adaptations needs to be updated only when interfaces are updated.
PIM Adaption Layer

- 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.

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 straightforward, but may require some complex logic.

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.
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 Translation
xtUML Modeling & Translation

xtUML Modeling & Testing
BridgePoint UML Suite

PIM
Modeled Test env. for Model Based Regression Tests

xtUML Model Translation
The /// Model Compiler

Marks
Model compilation flags for tuning of performance etc.

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)

xtUML Repository
xtUML Runtime (VM)

class
attribute
datatype
transition
state
event
instance
value
queue
stack

PSM Metamodel
Marking Model
Translation Rules
Design Patterns

xtUML Metamodel
xtUML VM

Translation Database
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
A new SW function was developed in multiple tracks:
- Hand-written C++ (synchronous design)
- Modelled using Executable UML (Shlaer-Mellor UML+AL)
- 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
Some UDPSH Class & State Diagr.
Model Compiler Enhancements

**Ericsson Proprietary Model Compiler**

- **AAL2 Pilot enhancements:**
  - Architectural optimizations (reduces memory costs)
  - Improved state machine scheduling (improves performance)
  - Support for Linux and nxtOSEK

- **Radical SW Pilot enhancements:**
  - Auto-alignment of data struct members (reduces memory costs)
  - Single job execution (reduces memory costs)
  - Architectural optimizations (reduces memory costs)
  - “Follow the leader” scheduling (improves performance)
  - Synchronous state machine execution (improves performance)
  - Support for Model Based Test Coverage
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.
### UDP SH 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**

![Graph showing CPU load for different approaches](image)

- Hand-coded C++: 7.0%
- BridgePoint UML+AL: 6.0%
- UML with C++: ? %

*not ready yet*
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

<table>
<thead>
<tr>
<th></th>
<th>Memory Utilization Stat &amp; Heap KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand-coded C++</td>
<td>901 KB</td>
</tr>
<tr>
<td>BridgePoint UML+AL</td>
<td>908 KB</td>
</tr>
<tr>
<td>UML with C++</td>
<td>? KB</td>
</tr>
</tbody>
</table>

Worst-case memory cost for these 10 test cases

not ready yet
HOT POTATO test: Sending data packets between state machines

Full tests was performed on different HW platforms & OS’es.
(PowerPC vs. Intel and OSE vs. Linux SW/HW platforms)
HOT POTATO signalling benchmark: BridgePoint vs. a UML with C++ 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)
BridgePoint runtime call graph

Proprietary xtUML Model Compiler tailor made for small footprint & highest performance.
UML with C++ tool runtime call graph

Off-the-shelf UML C++ code generator with adapted runtime system.
More of everything with Executable UML & MDA

("extra all" 😊 )
History of xtUML success stories

- **RBS: PP Adapter** – adapting existing RoseRT legacy to new HW platform
  - Developed by a trainee in parallel with an RBS hand-coding team.
  - Running all 30+ Use Cases on target in the Cello lab before the hand-coding team had written a single line of code. (= agile)

- **RNC: Load Control** – preventing overload situations in the RNC
  - Re-engineered by thesis students at RNC Design.
  - Worked on 1st attempt on target. ("has never happened before")
  - Benchmarked against the existing & optimized RoseRT solution.

- **DUPL: The AAL2 pilot**
  - Developed by senior engineers not programmed for ~15 years.
  - No faults on target.
  - Benchmarked against existing 10 year old & optimized solution.
    (load module sent to Croatia for testing)

- **PDU CPP: Radical SW pilot** – the UDPSH function
  - Worked on target right away.
  - Equal in performance and memory utilization with the newly hand-written C++ version.
  - Has been run on different HW with OSE and Linux.
    (as a UDPSH + TestBench monolith)
History of xtUML success stories

- "Surprisingly" high quality of software.
  - Typically no faults on target.
  - Coverity did not find any problems in generated code.

- Equal in performance with hand-optimized code.
  - Results from benchmarking of CPU load.

- Re-factoring is easier than expected.
  - Need a modelled test bench for re-testing the re-factored model.

- Full control of the entire code generation process.
  - Flexibility to alter code generation whenever needed.
  - Not dependent on tool vendor.
Conclusions

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

- It depends primarily on how good the Model Compiler is. (provided that the model not is ”badly” designed)

- We must learn to ”trust” & develop Model Compilers and learn how to develop & provide additional support & value for our proprietary platforms and reduce the costs for HW.

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

Albert Einstein