Model-level Execution

Preliminary Architecture Evaluation

István Gansperger, Róbert Kitlei, Gergely Dévai, Máté Karácsony,
Gábor Ferenc Kovács, Tamás Kozsik, Boldizsár Németh, Attila Ulbert

ELTE-Soft Nonprofit Ltd.

May, 2015

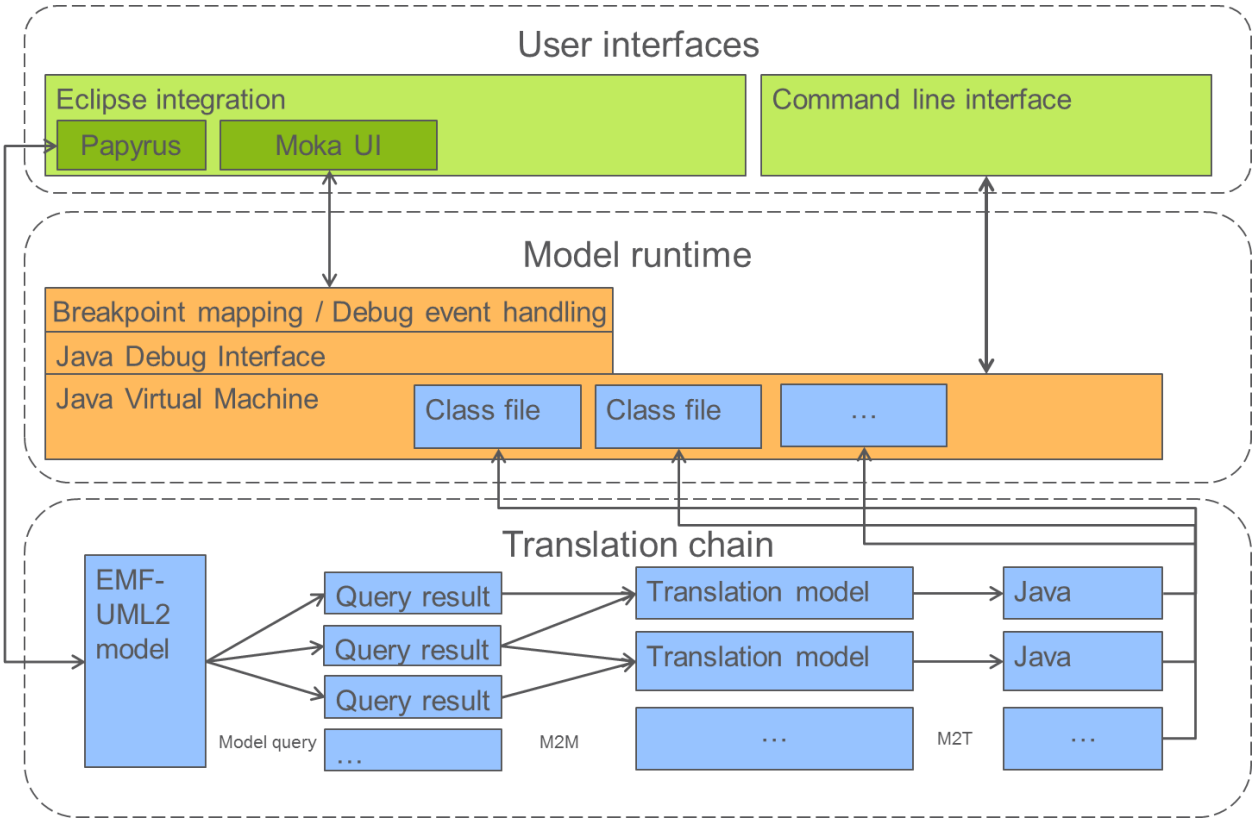
1 Contents

1 Introduction

The purpose of this document is to provide preliminary scalability results on the Model Executor architecture.

2 Architecture summary

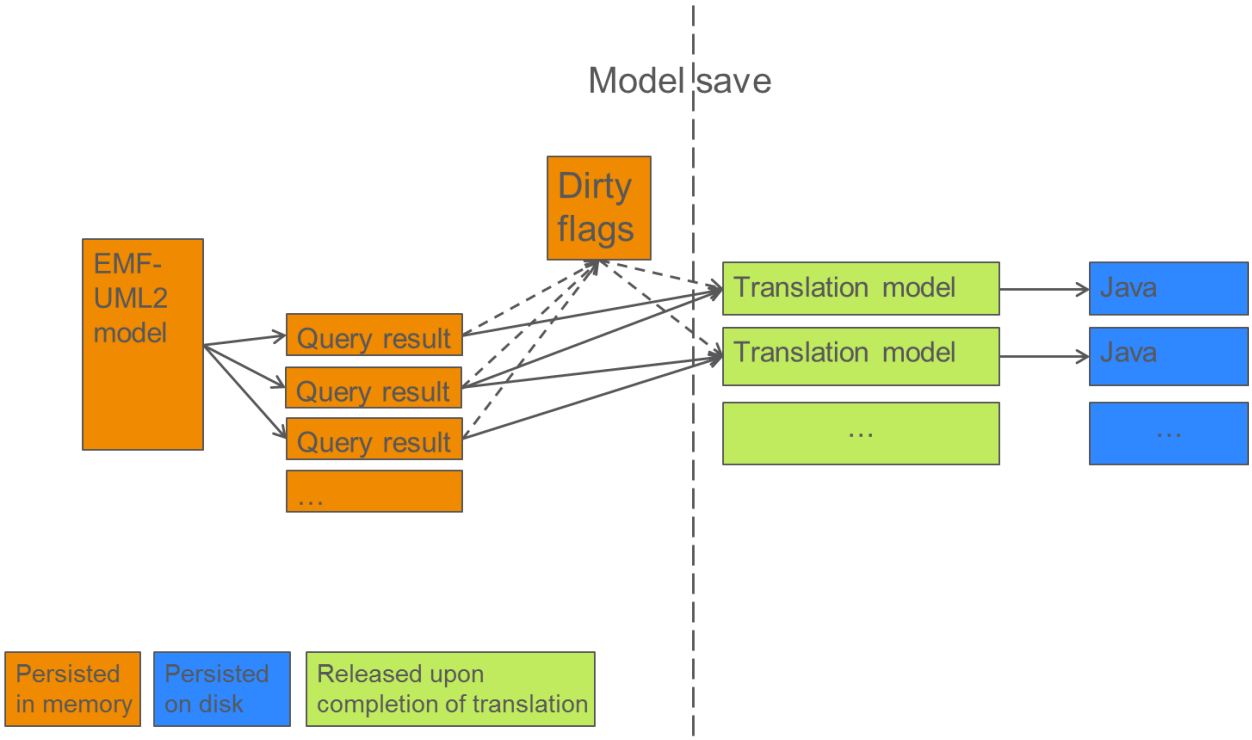
2.1 High-level view



The user creates EMF-UML2 models via Papyrus. The model changes are propagated through the translation chain to create Java source files that are compiled to class files. When model execution is initiated, a Java Virtual Machine is instantiated to load these class files in and to start execution. During interactive model execution, the user is exposed to the Moka user interface. Moka breakpoints and state machine animation is implemented by Java breakpoints that are set via the Java Debug Interface.

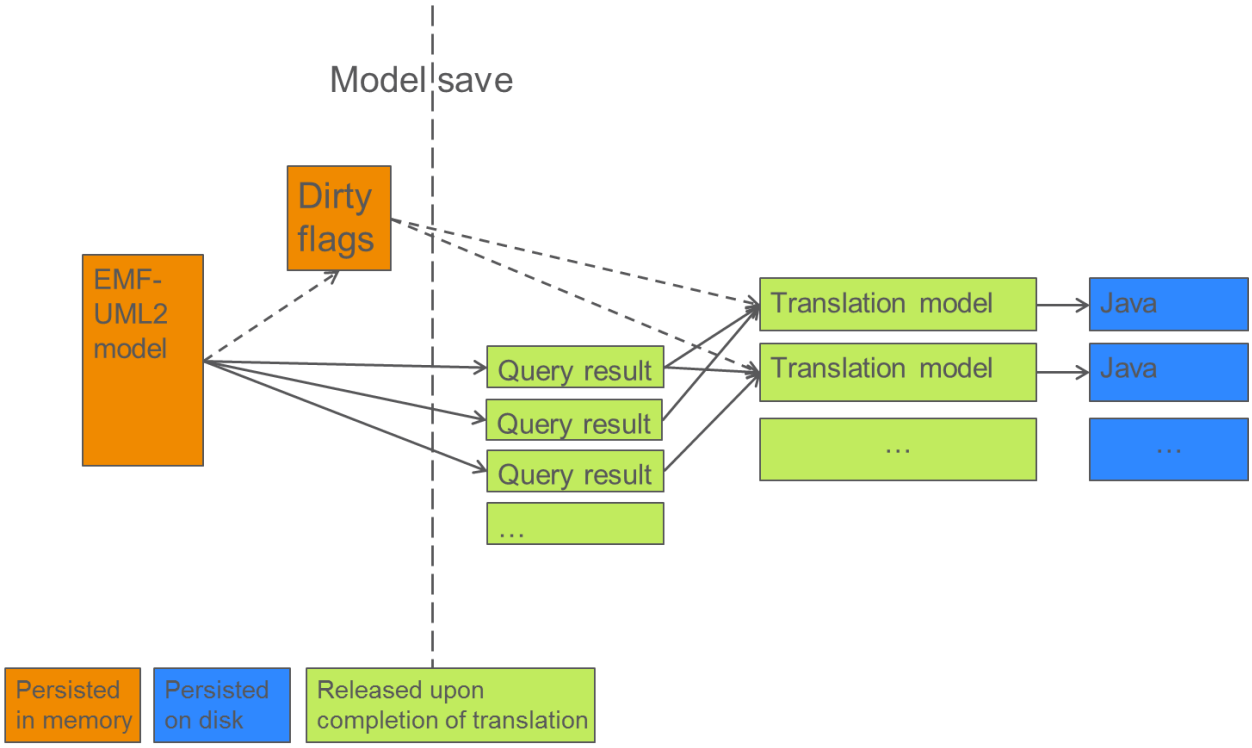
In addition to Eclipse integration, a command line interface is also provided that uses the same translation chain (but in non-incremental way) and runs the model without involving the Java Debug Interface.

2.2 Translation Chain with IncQuery



The model change events are propagated to IncQuery that updates the query results and creates change notifications. These notifications are handled by setting dirty flags (changed/new/deleted flags) on (to be) generated Java files. When translation is triggered (currently at model save), all dirty files are (re)generated by first building temporary translation models (on a per file basis) using the query results. The translation models are turned to Java text via Xtend templates and debug symbols are also generated.

2.3 Translation Chain with Java Queries



In this setup, the change notifications from the model are directly consumed by the module maintaining the dirty flags. On model save, all dirty files are regenerated by using model queries implemented in Java and the same translation models described in the previous section.

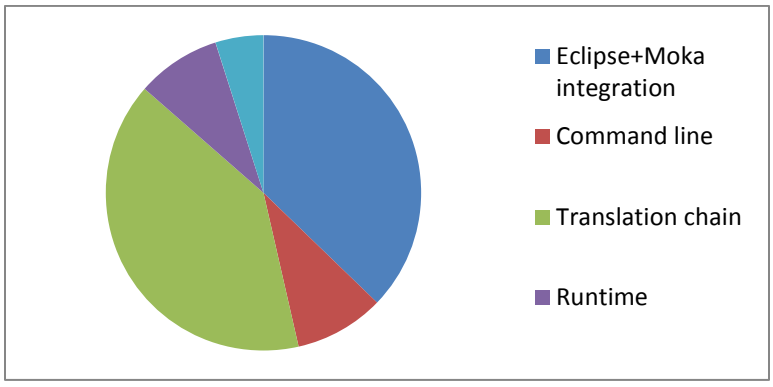
3 Code Size

Code size was measured by nonempty lines of code. Meta-models are currently small. Their sizes were measured by the number of XML lines to keep it simple and to increase their weight to express the intellectual work needed.

	Java	Xtend	IncQuery	Messages	Model	Plugin	SUM
3pp						66	66
cli	861			74			935
filemanager	192					10	202
ide	3832			170		177	4179
m2m.logic	1937					17	1954
m2m.metamodel					188	77	265
m2t.java	11	442				16	469
m2t.smap.emf	281					11	292
m2t.smap.xtend	439	209				13	661
runtime	864					12	876
uml.alf	48	80				11	139
uml.incquery	12		134			34	180
SUM	8477	731	134	244	188	444	

Breaking down the sums according to the main architectural modules, we get the following figures:

Eclipse+Moka integration	3779
Command line	935
Translation chain	4062
Runtime	876
Debug symbol generation	500



4 Memory Consumption Experiments

The memory consumption experiments described in this section were run on a standard Ericsson laptop with Windows 7. The Model Executor plugin was installed in Eclipse (rather than starting an inner eclipse) to be able to simulate end-user environment. Memory consumption measurements are based on what the operating system reported on the JVM running Eclipse, to be non-intrusive.

4.1 Project cost

An Eclipse instance with zero opened projects costs ~300 MB in the configuration used. The following table shows that the xUML-RT project causes ~85 MB overhead compared to a Papyrus project. This is due to the Java nature (~60 MB) and the executor runtime JARs (~25 MB).

	MB
Papyrus project	326
Java project	388
Java project with executor runtime JARs	413
xUML-RT project	411

4.2 Translation cost

The following table shows memory consumption at different stages of using a project, in case of xUML-RT project with or without IncQuery. The Java project, in which case the generated Java sources of the previous two projects are added manually, is used as a reference to see how much of the used memory actually belongs to Java compilation.

Each of the models that are copied into and deleted from the model contains a single class with a state machine of 9, 100 and 1000 states and the same number of transitions respectively. The memory consumption of most of the steps is measured after it became stable after garbage collection, except the translation steps, where the peak memory consumption is relevant.

Action	xUML-RT project - IncQuery (MB)	xUML-RT project - Java query (MB)	Java project with Java code manually added (MB)
open Eclipse (stable)	306	309	307
open project (stable)	411	410	413
translation sm9 (peak)	491	485	459
delete sm9 (stable)	468	461	423
translation sm100 (peak)	495	484	464
delete sm100 (stable)	477	468	425

translation sm1000 (peak)	524	511	461
delete sm1000 (stable)	492	481	434

The results show that the memory cost of the translation is varies between 75-115 MB, of which 45-50 MB is consumed by Java compilation and the rest is due to the model-to-Java translation. The cost of IncQuery in this setup was 5-15 MB.

5 Runtime Experiments

The following table shows running times in milliseconds of the translation of state machines of different sizes. The first one contains 9 states, 9 transitions and an entry action with 9 send statements. The other two state machines are of the same structure with 100-100-100 and 1000-1000-1000 elements respectively. The model executor version with IncQuery was used for this experiment.

	Java gen	**Full**		**Java gen**	**Full**		**Java gen**	**Full**
9 states			**100 states**			**1000 states**		
copy model	4165	-	copy model	968	-	copy model	4024	-
rebuild	968	1030	rebuild	951	1014	rebuild	3183	3245
rebuild	811	889	rebuild	764	842	rebuild	3556	3603
rebuild	499	561	rebuild	608	655	rebuild	3869	3931
rebuild	406	453	rebuild	624	687	rebuild	2808	2854
change state	18283	18283	change state	983	1014	change state	4442	4489
change state	639	655	change state	1264	1280	change state	4973	4988
change state	671	702	change state	702	733	change state	5880	5895
change action	639	655	change action	702	718	change action	1669	1685
change action	609	640	change action	702	718	change action	1232	1248
change action	780	858	change action	514	530	change action	1669	1669

The “Java gen” column shows the running times from the startup of the translation process (events triggered by saving the model or a project build) to the last Java file written on the disk. In addition to this, the “Full” column also includes the time of compilation to class files and “patching” those with debugging symbols. There is only a small difference between the two columns. The reason for this is that JDT starts the Java compilation on separate threads as soon as the first generated Java file appears.

The execution times are measured throughout the following actions:

- “copy model”: Add the model into the empty project and translate it for the first time. (The “full” column is empty here due to a bug in the executor version used for the measurements.)
- “rebuild”: Clean and rebuild of the project.
- “change state”: Changing the name of the state. This triggers the re-generation of the full region, so this is almost equivalent to full rebuild.
- “change action”: A (new) entry action, consisting of only a couple of send instructions, is modified. This triggers the regeneration of a single, small Java file.