# fUML as an Assembly Language for MDA

Frédéric Jouault
TRAME team, ESEO
Angers, France
frederic.jouault@eseo.fr

Massimo Tisi
AtlanMod, École des Mines de Nantes - INRIA, LINA
Nantes, France
massimo.tisi@inria.fr

Jérôme Delatour
TRAME team, ESEO
Angers, France
jerome.delatour@eseo.fr

## ABSTRACT

Within a given modeling platform, modeling tools interoperate efficiently. They are generally written in the same general purpose language, and use a single modeling framewok (i.e., an API to access models). However, interoperability between tools from different modeling platforms is much more problematic.

In this paper, we argue that fUML may be leveraged to address this issue by providing a common execution language, and by abstracting modeling frameworks into generic actions that perform elementary operations on models. Not only can user models benefit from a unified execution semantics, but modeling tools can too.

## Keywords

fUML, MDA, Tool Interoperability

## 1. INTRODUCTION

A modeling platform (e.g., Eclipse Modeling) consists of a set of modeling tools (e.g., constraint checkers, comparators, transformation engines) that can be used together. Interoperability between tools of a given platform is typically achieved by two means: 1) a common programming language (e.g., Java), and 2) a common modeling framework (e.g., EMF: Eclipse Modeling Framework).

Interoperability across modeling platforms generally relies on a common interchange format (e.g., a given version of XMI) to exchange models. However, modeling tools cannot be exchanged (i.e., ported) so easily between platforms that rely on different programming languages or modeling frameworks. Therefore, some tools are either not available on some platforms, or have multiple implementations that are possibly inconsistent.

Moreover, some tools are actually execution engines for modeling languages. Such tools implement the semantics of modeling languages. For instance, an OCL constraint evaluator implements the semantics of OCL. Therefore, exchanging semantics implemented in tools is as problematic as exchanging tools.

In this paper, we argue that a common modeling Virtual Machine (VM) may be used to implement a variety of modeling tools. We call *modeling VM* a virtual machine that abstracts the specifics of modeling platforms: programming language, and modeling framework. Tools built on top of this VM, or compiled to this VM, become portable across all modeling platforms providing an implementation of the VM.

We then consider the requirements for such a VM in the context of Model-Driven Architecture (MDA). Although the idea of a modeling VM can be discussed in the broader Model-Driven Engineering (MDE), we decide here to restrict ourselves to MDA. Indeed, as a set of standards, MDA could benefit from covering this aspect as well.

Then, we proceed to show that fUML could be used as a modeling VM for MDA. This application of fUML is different from its typical usage scenario. Instead of only using fUML to specify the behavior of models created by users of a modeling platform, we propose to also use it to provide execution for modeling tools. We are thus considering the use of models, expressed in fUML, at the runtime of modeling tools. When used in this way, fUML shares some characteristics with assembly languages: it is only **rarely used directly**, and complex fUML models are only **rarely displayed in a readable way**. The fUML specification recognizes these points, and defines a textual language to represent fUML activities.

We also show that, however, making fUML actually usable in this way requires to modify it. Some modifications, like adding support for interruptions, can also benefit users who model systems in fUML, but others are specifically intended for its use as a modeling VM assembly language. Therefore, the purpose of this paper is not to impose fUML in this role, but simply to position it as a possible assembly language for MDA, and to express our interest in exploring this idea further.

The analogy of fUML to an assembly language is similar (and inspired by) the well-known analogy "Javascript is assembly language for the Web" [1, 2]. Obviously, this is only an analogy that has its limits. For instance, fUML (like Javascript) is at a higher-level of abstraction than assembly languages typically are. However, we strongly believe that this analogy may be as beneficial to MDA as the analogy about Javascript is to the Web.

The paper is organized as follows: Section 2 motivates the need for a modeling Virtual Machine. The requirements for


*WOODSTOCK* '97 El Paso, Texas USA

such a VM in the MDA context are presented in Section 3. Based on these requirements, Section 4 discusses why fUML may be such a VM for MDA. Finally, we conclude in Section 5.

## 2. MOTIVATION

In this section, we motivate the need for a modeling Virtual Machine (VM). This VM may be used as an execution platform for modeling tools. In the remaining of the paper, we refer to the language that this VM executes as *VM language*. For instance, Java bytecodes form the VM language of the Java virtual machine.

A classical VM such as the Java VM provides abstractions for: 1) actual hardware (e.g., processor, memory, peripherals), and 2) operating system (e.g., Windows, Linux). This makes tools running on top of a classical VM independent of hardware and operating system. A modeling VM additionally provides abstractions for: 3) modeling framework (e.g., EMF), and 4) the general-purpose language in which the modeling platform is implemented (e.g., Java, C#, C++). Therefore, tools running on top of a modeling VM are also independent of how the modeling platform is implemented.

With a modeling VM, it becomes possible to develop modeling tools that can run on any modeling platform. Moreover, certain kinds of optimizations may be performed on modeling VMs independently of tools.

A modeling VM may be used to provide execution to a wide variety of modeling tools such as: simulator, animator, debugger, comparator (diff, merge), version control, importer, exporter, constraint checker, or formal checker. Some modeling tools are actually execution engines for modeling languages used to specify: transformations (model to model, model to text, or text to model), constraints, etc. There are two main approaches to implement an execution engine: translation of programs to the VM language (i.e., compilation), or interpretation of programs by an interpreter expressed in the VM language.

Another significant application for a modeling VM is the implementation of standards. For instance, the MDA standards from the OMG[1] may be implemented on top of a modeling VM:

- **Execution engines for languages**: Object Constraint Language (OCL), fUML , MOF Model to Text Transformation Language (MOFM2T), and Query / View / Transformation (QVT).
- **Importer and exporter** for XML Metadata Interchange (XMI), Action Language for Foundational UML (Alf), and Human-Usable Textual Notation (HUTN).

If a modeling VM is standardized, it may make sense to provide reference implementations for other standards on top of it.

## 3. REQUIREMENTS

Among the plethora of languages that have been defined or simply used by the MDE community, several may be considered as candidates for the role of assembly language. At the time of writing this paper, the Java+EMF combination is widely used by MDE tools, and can very well be considered as an MDE VM. However, having decided to consider the issue from the MDA perspective, we must express our requirements accordingly.

In this section we enumerate a set of requirements for an MDA assembly language, distinguishing mandatory requirements from optional ones.

The set of mandatory requirements includes the main expected properties for an assembly language for MDA:

- **MDA standard.** The VM language should be recognized as a standard, and its formal semantics should be publicly available. This requirement appears first because MDA is our scope. Obviously, such a requirement would not be mandatory for an MDE VM.
- **Computational completeness.** Using the VM language it must be possible to specify any possible model operation, i.e., any computable function on models.

Other requirements, despite not being mandatory, would increase the value of the VM language:

- **Model handling.** Models and model elements should be manipulated as first-class entities by the language, without the need to encode them into other data structures.
- **No over-specification.** When translating other languages to it, the VM language should not impose to specify unneeded information. For instance, the VM language should allow programs to specify sequencing between instructions only if necessary, allowing for implicit parallelism (this property is especially significant now that virtually every computer is multicore). Analogously the VM language should not impose an order for side-effect free evaluations, thus allowing for eager or lazy evaluation of a given program when needed.
- **Exception handling.** Language constructs for exception handling at the VM level are not mandatory, but would simplify the use of the VM language.
- **Introspection and Reflection.** When a model-driven tool uses introspection and reflection, its implementation over the VM is made easier if this support is embedded in the VM language. For instance, reflective access to model elements is a commonly used feature in MDE tools.
- **Modularity/Composability.** It should be possible to compile parts of tools into separate modules to be composed in the VM language.
- **VM code as a model.** For uniformity with the development platform, VM code should be represented in the form of a model. Model interchange mechanisms (e.g., XMI) can then be leveraged for VM code as well. This would also allow to manipulate VM code using VM language. This is similar to higher-order transformations [3], and may be leveraged to use model transformation to specify compilers that target the VM language.
- **High-performance implementation.** It should be possible to build a high-performance implementation of the language, thus reducing the performance penalty of using a VM.

[1] As listed at: `http://www.omg.org/spec/`.

- **Wide availability.** The language and its implementation should be publicly available, and widely used in the community.

At this stage, it should be noted that conflicts between optional requirements cannot be completely resolved. For instance, **high-performance implementation** may rely on static computation of a control flow, which may be seen as contradicting **no over-specification**. An answer to these requirements will be a trade-off.

Finally, there are properties that are typically valuable for a language in MDE, but that have no primary importance for a VM language, such as: human-readability, conciseness, and maintainability[2].

Because fUML is the only MDA standard at the right level of abstraction to play the role of VM language, we have no other MDA point of comparison. Therefore, we compare fUML to non-MDA modeling VMs such as Java (e.g., used with EMF), and ATL VM in order to see which aspects of fUML are adequat, and which may need to be improved. Table 1 gives an overview with the set of requirements presented above as rows, and languages under consideration as columns. The set of languages we take into consideration is: 1) the Java VM language, a general-purpose VM language currently targeted by several MDE tools, especially in the Eclipse modeling platform, 2) the ATL VM language[3], a high-level VM for model manipulation, 3) the fUML language version 1.1, Beta 1 [5]. This set is not meant to be exhaustive, as several other languages could be included in this comparison, such as: Kermeta [6], or Epsilon [4]. However the three languages considered are representative of VM languages at different abstraction levels.

# 4. FUML AS ASSEMBLY

In this section we argue that fUML is an interesting candidate to be an assembly language for MDA: it satisfies many of the identified requirements, and is expected it to satisfy most of the others.

As it is shown in Table 1, the requirement of computational completeness is satisfied by all the three languages under consideration. Instead only one of the three VMs, the ATL VM, natively handles model elements as first-class entities. This requirement is in general satisfied by VMs that, like the ATL VM, are designed for MDE. However general purpose VMs like the Java VM can still provide a uniform model access by using common modeling frameworks like EMF. On the other hand, fUML is designed to handle exclusively instances of UML classifiers but it can be generalized to any model element by lifting its semantics to MOF, similarly to what has been done by xMOF [7].

Among the three VMs, the mature Java and ATL VMs provide a high-performance implementation, but a similarly efficient machine is expected also for fUML. Finally, while fUML is today the least popular VM among the three choices, we expect it to become widely available once the standard will be mature.

[2]For instance, Java bytecode is barely readable (especially in binary format), not especially concise, and generally not maintained directly.

[3]We refer to the most recent version of the ATL VM, named EMFTVM [4], unless specified otherwise in a cell of Table 1.

[4]`http://www.eclipse.org/epsilon/`

There are three rows of Table 1 that illustrate which control is provided by the VM language over code execution. In particular we focus on the possibility of specifying that 1) some operations may be executed sequentially or in parallel, 2) evaluations may be performed in a lazy or eager way, 3) operations may be executed without a specific execution order. The Java VM imposes a sequential order between its instructions, but allows for explicitly defined parallel operations or lazy evaluations by relying on programming libraries written in VM bytecode. We call this approach *in-language* in Table 1. Also the ATL VM bytecode is used in fixed sequences of imperative instructions, and its current implementation does not allow to specify parallelism or laziness. Finally in fUML parallelism is a *linguistic* feature of the language, that allows the definition of regions of parallel execution. fUML does not impose an execution order between instructions, as it provides a specific dataflow semantics for edges. Lazy evaluation is not explicitly supported by fUML linguistic features but can be still implemented by specific modeling patterns. Moreover, UML activity diagrams have a linguistic support for lazy evaluation (with ValueInputPin and ActionPin) and an extension of fUML in this sense would be possible.

Advanced features like exception handling and reflection are only supported by the Java VM. All VM languages implement at least a modularity/composability mechanism. With respect to the other two VMs, the fUML option has the important benefits of representing VM code as a model and of being an MDA standard.

With respect to the use of a modeling VM to implement standards (see discussion at the end of Section 2), it should be noted that the fUML standard already comes with a reference implementation of fUML specified in fUML.

As an example of roadmap that MDE tools may follow to comply to a common modeling VM, we discuss the case of the ATL transformation language. A possible roadmap may involve replacing the current version of the ATL compiler (that compiles towards the ATL VM) with a new compiler towards fUML, encoding transformation rules as flows of fUML activities. fUML activities, lifted to the MOF level, would directly modify the models under transformation. An immediate benefit of this new compiler w.r.t. the old one would be the possibility of leveraging the innate parallelism of fUML by exploiting fUML parallel execution regions in the generation. When semantically-equivalent implementations of fUML in different modeling platforms will be available, the same ATL transformation would be executable in any of these platforms.

# 5. CONCLUSION

In this work, we explained that interoperability between modeling tools across modeling platforms may be simplified by the use of a common modeling Virtual Machine (VM). This VM provides execution to modeling tools written in (or compiled to) its language.

Although fUML is not fully ready to be the language for such a modeling VM, it is one of the best candidates. Once the problems mentioned in Section 4 are addressed, it does not lack many features to be usable for many kinds of tools. However, we have not considered tools that have a stronger dependency to a given platform. For instance, graphical model editors generally have a strong dependency to a windowing toolkit, which is relatively complex to abstract in a

**Table 1: Languages and requirements.**

| | Java VM bytecode | ATL VM bytecode | fUML |
|---|---|---|---|
| **Computational completeness** | yes | yes | yes |
| **Model handling** | only by using modeling framework | yes | only UML InstanceSpecifications (but can be lifted to MOF) |
| **High-performance implementation** | yes | yes | no (but expected) |
| **Wide availability** | ++ | + | - (++ expected) |
| **Parallel operations** | in-language | no (but linguistic in ParallelVM[a]) | linguistic |
| **Eager/Lazy evaluation** | in-language | no (but linguistic in LazyVM [8]) | in-language (but linguistic may be added, as in UML activities) |
| **No execution order over-specification** | fixed execution order | fixed execution order | execution order not imposed |
| **Exception handling** | yes | no | no (but it may be added, as in UML activities) |
| **Introspection and Reflection** | yes | only reflective model access | no |
| **Modularity/ Composability** | yes | yes | yes |
| **VM code as a model** | no | yes | yes |
| **MDA Standard** | no | no | yes |

[a] `http://www.emn.fr/z-info/atlanmod/index.php/Parallel_ATL`

VM.

Execution performance of fUML may be limited on full-fledged fUML engines, which provide complete simulation of the flow of tokens. These tools (e.g., [9]) are extremely useful when fUML is used to specify behavior of user models. However, when fUML is used as a VM, techniques similar to those used for `asm.js` [10] may be used to increase performance: definition of a simpler subset of fUML, and ahead-of-time compilation to machine code. Therefore, performance should not be an issue that prevents using fUML as proposed in this paper.

# 6. REFERENCES


[1] Kappe, D.: Is Javascript the Assembly Language of Web 2.0? `http://pathfindersoftware.com/2007/03/is_javascript_t/`. Accessed: 2013-07-22. (Archived by WebCite® at `http://www.webcitation.org/6IIxYG22S`) (March 2007)

[2] Hanselman, S., Meijer, E.: JavaScript is Assembly Language for the Web: Semantic Markup is Dead! Clean vs. Machine-coded HTML. `http://www.hanselminutes.com/274/javascript-is-assembly-language-for-the-web-semantic-markup-is-dead-clean-vs-machine-coded`. Accessed: 2013-07-22. (Archived by WebCite® at `http://www.webcitation.org/6IIz8ZvNt`) (July 2011)

[3] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Model Driven Architecture-Foundations and Applications, Springer (2009) 18–33

[4] Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: Model Driven Engineering Languages and Systems. Springer (2011) 623–637

[5] (OMG), O.M.G.: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.1, Beta 1. `http://www.omg.org/spec/FUML/1.1/` (October 2012)

[6] Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems. Volume 3713 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 264–278

[7] Mayerhofer, T., Langer, P., Wimmer, M.: Towards xMOF: executable DSMLs based on fUML. In: Proceedings of the 2012 workshop on Domain-specific modeling. DSM '12, New York, NY, USA, ACM (2012) 1–6

[8] Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: Model Driven Engineering Languages and Systems. Springer (2011) 32–46

[9] Mayerhofer, T., Langer, P., Kappel, G.: A runtime model for fUML. In: Proceedings of the 7th Workshop on Models@run.time. MRT '12, New York, NY, USA, ACM (2012) 53–58

[10] Herman, D., Wagner, L., Zakai, A.: asm.js. `http://asmjs.org/spec/latest/` (March 2013)