An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements

François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel

An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements

François Fouquet[1], Grégory Nain[2], Brice Morin[3], Erwan Daubert[1], Olivier Barais[1] Noël Plouzeau[1], and Jean-Marc Jézéquel[1]

[1] University of Rennes 1, IRISA, INRIA Centre Rennes
Campus de Beaulieu, 35042 Rennes, France
{Firstname.Name}@inria.fr
[2] SnT - University of Luxembourg
Luxembourg, Luxembourg
{gregory.nain}@uni.lu
[3] SINTEF
Oslo, Norway
{Brice.Morin}@sintef.no

Abstract. Models@Runtime aims at taming the complexity of software dynamic adaptation by pushing further the idea of reflection and considering the reflection layer as a first-class modeling space. A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework. EMF provides facilities for building DSLs and tools based on a structured data model, with tight integration with the Eclipse IDE. EMF has rapidly become the defacto standard in the MDE community and has also been adopted for building Models@Runtime platforms. For example, Frascati (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities. This paper highlights specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative implementation to meet these requirements.

Keywords: Model@Runtime, EMF, adaptation

1 Introduction

The emergence of new classes of systems that are complex, inevitably distributed, and that operate in heterogeneous and rapidly changing environments raise new challenges for the Software Engineering community [3]. Examples of such applications include those from crisis management, health-care and smart grids. These applications can be deployed on top of a distributed infrastructure that goes from micro-controller to the Cloud. These systems must be adaptable, flexible, reconfigurable and, increasingly, self-managing [9]. Such characteristics make systems more prone to failure when executing and thus the development and

study of appropriate mechanisms for continuous design and runtime validation and monitoring are needed. In the Model-Driven Software Development area, research effort has focused primarily on using models at design, implementation, and deployment stages of development. This work has been highly productive with several techniques now entering the commercialization phase. The use of model-driven techniques for validating and monitoring run-time behavior can also yield significant benefits. A key benefit is that models can be used to provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns such as verification and monitoring. Then, Models@Runtime [2] denotes model-driven approaches aiming at taming the complexity of software and system dynamic adaptation. It basically pushes the idea of reflection [11] one step further by considering the reflection layer as a real model: "*something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality*" [14], which enables the continuous design of complex, adaptive systems.

A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework (EMF) [4]. For example, Frascati [16] (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities and its use to support Models@Runtime reaches some limitations. This paper elicits specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative modelling framework implementation to meet these requirements.

The outline of this paper is the following. Section 2 briefly presents the Models@Runtime paradigm and its requirements. An overview of EMF benefits and its limitations regarding its use at runtime are given by Section 3. The contribution of this paper, the Kevoree Modeling Framework(KMF), is described in Section 4. Section 4.2 gives an evaluation of our alternative implementation in comparison to EMF. This contribution is discussed *w.r.t.* related work in Section 5. Section 6 concludes on about this work and presents future work.

2 Models@Runtime Requirements

The Models@Runtime paradigm promises a new approach to MDE, by fading the boundary between design-time (the typical phase where MDE is employed) and runtime. More precisely, the goal of Models@Runtime is to enable the continuous design, evolution, verification of eternal running software systems [2]. A typical usage of Models@Runtime is to manage the complexity of dynamic adaptation or verification in complex, **distributed** and **heterogeneous** systems, by offering a more abstract and safer abstraction layer on top of the running system than reflection. Heterogeneity and distribution creates specific requirements for Models@Runtime infrastructure. (i) The overhead inevitably induced by this

[4] http://www.eclipse.org/emf/

advanced reflection layer should not prevent smaller (*i.e.* resource constrained) devices to benefit from the advantages of Models@Runtime (e.g. Java Embedded, Android,...). Modeling framework and all its needed dependencies must be compatible with such devices in terms of memory footprint. (ii) The use of models to drive the running configuration of a software system should enable required features of a distributed reflection layer such as efficient (un)-marshalling, efficient model cloning and model thread safety access.

2.1 Reduced Memory Footprints

The memory footprint of a Models@Runtime engine basically determines the types of nodes able to run this engine. The more demanding is the Models@Runtime engine in terms of memory, the more difficult it is to deploy it on the smallest devices (e.g. Android phones, gateways with low power CPUs), and the more centralized should the adaptation/verification be. Lazy loading technique can be used to virtually reduce the memory overhead by not loading unused model elements. In this case, only a few large devices would be able to reason and make decisions for all the smaller devices. This would reduce the reliability of the overall adaptation and verification process: if the large devices fail, the overall system cannot safely adapt anymore. Moreover, model exchanges for the synchronization of the system in this strategy would dramatically increase network load.

2.2 Dependencies

The number and size of dependencies is also an important criteria. Each device must provision all the dependencies needed by the modeling framework to run a Models@Runtime based distributed application. As these applications are based on a structured data model, this data model should not generate useless dependencies. Heavy dependencies would indeed increase the time needed to initialize a node or update it when new versions of those third parties are available.

2.3 Thread Safety

A Models@Runtime is generally used in highly concurrent environment. For instance, different probes integrated in a device update a context model. This model is then used for triggering the adaptation reasoning process. This context model should enable safe and consistent read and write for the reasoners to take accurate decisions. The Models@Runtime infrastructure must ensure that the multiple threads of your application can access and modify the models without worrying about the concurrent access details. In particular, it should be possible to navigate in parallel the collections defined in the model to implement fast, yet safe, validation or reasoning algorithms on multi-core/thread nodes.

2.4 Efficient Model (Un)Marshalling and cloning

A device should be able to locally clone its own model for verification or reasoning purposes so that it can reason on a fully independent and safe representation of itself, which can later on be re-synchronized with the current model. Also, devices should be offered efficient means to communicate their Models@Runtime to neighbors so that collective decisions can be made. The Models@Runtime infrastructure must thus provide efficient model cloning and (un)marshalling capabilities.

2.5 Connecting Model@Runtime to classical design tools

This requirement is directly bound to the first goal of fading the boundary between design-time and runtime. A Models@Runtime infrastructure must provide a transparent compatibility with design environments. For example, a graphical simulator used for the design of finite state machines (FSM) should be pluggable on an application that keeps FSM at runtime and serve as a debugger or a monitor of the running system [1,7].

3 EMF Benefits and Limitations

A natural way to implement a Models@Runtime platform is to rely on tools and techniques well established in the MDE community, and in particular, the *de facto* EMF standard. This section provides a brief overview of EMF and then discusses the suitability of this modelling framework with respect to the requirements identified in the previous section.

3.1 EMF Overview

EMF is an eMOF implementation and code generation facility for building tools and other applications based on a structured data model. From a model specification, EMF provides tools and runtime support to create Domain Specific Language (DSL) on top of the Eclipse platform. Most important, EMF provides the foundation for interoperability with other EMF-based tools and applications using a default serialization strategy based on XMI. Consequently, EMF has been used to implement a large set of tools and thus, evolved into an efficient Java implementation of a core subset of the MOF API.

3.2 Advantages.

As a first real benefit, EMF provides a transparent compatibility of the Models@Runtime infrastructure with several design environments. All the tools built with frameworks such as Xtext [10,4], EMFText [8], GMF [15] or ObeoDesigner [5]

[5] http://www.obeodesigner.com/

can be directly plugged on the Models@Runtime infrastructure to monitor the running system. The generated code is clean and provides an embedded visitor pattern and an observer pattern [6]. EMF also provides an XMI marshaller and unmarshaller that can be used to easily share models. Finally EMF offers lazy loadings of resources allowing the loading of single model elements on demand and caching them softly in an application.

3.3 Limitations.

To highlights the limitations of EMF, we will use the following experiment based on a simple Finite State Machine (FSM) metamodel with four meta-classes (FSM, State, Transition and Action in Fig. 1). A FSM contains the States of the machine and references to initial, current and final states. Each State contains its outgoing Transitions, and Transitions can contain Actions. From this tiny example, we discuss thread safety and dependencies, and we evaluate the memory footprint, as well as model (un)marshaling and cloning.

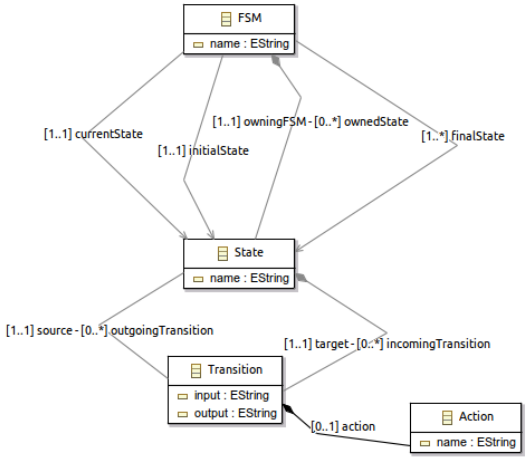

Fig. 1. Finite State Machine Metamodel used for Experiments

Large dependency set. Figure 2 shows the plugin/bundle dependencies for each new EMF generated code. By analyzing these dependencies one can see that the generated code is tightly coupled to the Eclipse environment and to the Equinox runtime (Equinox is the version of OSGi by the Eclipse foundation). Although this is not problematic when the data model is integrated as an Eclipse plugin (with all dependencies imposed by the Eclipse environment); these dependencies are more difficult to resolve and provision when this metamodel is used outside Eclipse, *i.e.* in a standalone context.

For the simple FSM metamodel, a standalone JAR executable outside of the Eclipse tool (a Java archive that including all dependencies) has a size of **15 MB** for only **55 KB** generated files. This footprint is rather difficult to reduce with tools like ProGuard[6], since it contains a large number of reflexive calls, which could potentially and implicitly affect any code in these 15 MB.

The large number and size of dependencies is one of the main limitations of EMF when the model must be embedded at runtime.

Static registries and multi-class loader incompatibility. Many runtime for dynamic architecture (e.g. OSGi, Frascati, or Kevoree) need to use their own class loader to properly manage and improve dynamic class loading. Consequently, the second limitation comes from the use of static registries in EMF, that leads to incompatibilities with runtime using multi-class loaders.

Lack of thread safe access to the models. EMF does not provide thread safe accesses to the models [7]. This requirement is important for a Models@Runtime infrastructure, because the support for dynamic and distributed architectures requires concurrent access to models.

Cloning overhead. Another limitation is the large memory footprint of marshaling, unmarshaling and cloning in the EMF implementations. To measure this limitation, we programmatically created a model with 100,000 State instances, with a transition between each state and an action for each Transaction. The results for EMF are the following.
On a Dell Precision E6400 with a 2.5 GHz iCore I7 and 16 GB of memory, the model creation lasts 376 ms, its marshaling to a file lasts 7021 ms and uses 104 MB of heap memory. The cloning using `EcoreUtil` lasts 3588 ms, and loading the model from a file lasts 5868 ms[8].

3.4 Synthesis.

Table 1 summarizes the advantages and limitations for the usage of EMF as a foundation for a Models@Runtime infrastructure. For each criterion, we put a $\vee$ when EMF provides advantages, $\times$ when we see limitations of using EMF for building Models@Runtime infrastructure, $\sim$ when we see possible improvements.

[6] ProGuard is a code shrinker (among other features not relevant for this paper) for Java bytecode: http://proguard.sourceforge.net/
[7] http://wiki.eclipse.org/EMF/FAQ#Is_EMF_thread-safe.3F
[8] This experiment can be downloaded http://goo.gl/CyLLC

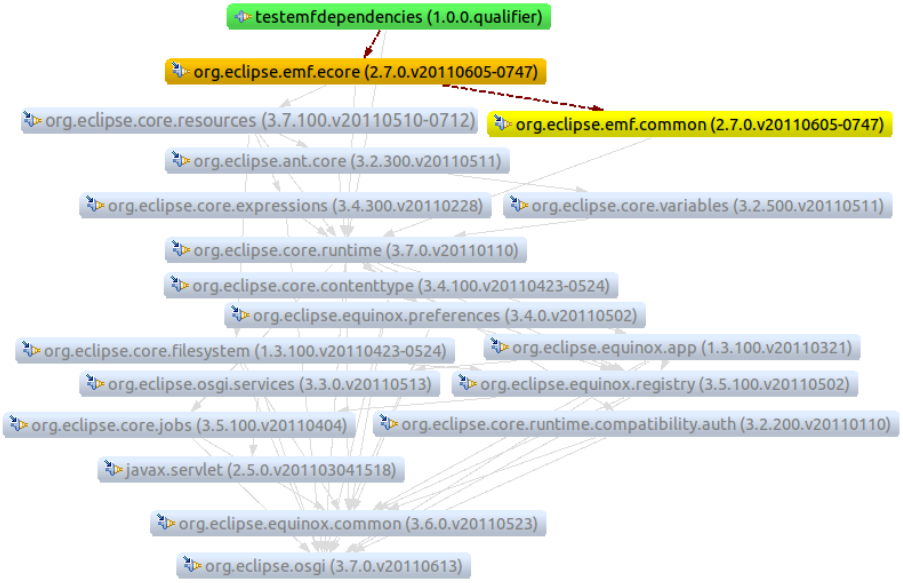

Fig. 2. Dependencies for each new metamodel generated code

Memory footprints	×
Lazy loading	∨
Dependencies	×
Thread safety	×
Efficient model (un)marshalling and cloning	∼
Connecting design tools	∨

Table 1. EMF features compared with Models@Runtime requirements

4 Kevoree Modeling Framework

KMF, or Kevoree Modeling Framework [9], is our alternative realization of EMF, which was formerly developed as part of our Kevoree Models@Runtime engine. This section presents the design choices we made to support a generic and efficient Models@Runtime infrastructure compatible with EMF. The general idea of KMF is threefold:

1. KMF aims at keeping the compatibility with EMF to guarantee the compatibility with design environment and the marshalling and unmarchalling of models.

[9] https://github.com/dukeboard/kevoree-modeling-framework

2. KMF aims at leveraging the powerful features provided by modern programming languages (here, Scala) [13] to provide a proper design to handle models.
3. KMF aims at providing a generic Models@Runtime infrastructure to ease the heterogeneity and the distribution management.

4.1 Model handling

Regarding the Table 1, KMF provides the same features than EMF for code generation facilities and models (un)marshalling. All the generated artefacts are written in Scala. Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented (Traits) and functional languages, and provides bytecode compatibility with Java [13]. Scala uses type inference to combine static safety with the concise syntax of dynamically typed languages. The Scala features particularly relevant for KMF are the following: ByteCode compatibility with Java libraries, concept of traits, concept of *Option*, XML embedded, immutable List, concept of closure and efficiency.

Domain classes are generated as a set of Scala traits to ease the support of multiple inheritance and meta-model extension [5]. Traits are seen from Java Code as a Type. They can only be initialized through the generated Factory (as in EMF). Note that the generated Traits do not inherit from EMF `EObject` and that all references are initialized. In particular, collections are initialized and references to single objects with a lower bound of 0 rely on Options. These Scala options are a neater way to deal with null pointers[10]. Indeed, Option does not save the developer from ever having null, but that developer can only get null when he wants it. If it is semantically impossible for a value to be null, the type checker enforces it. The getters on collection use immutable lists. The generated code provides helpers to add and remove model elements on collections, and it also provides specific methods to ease mixed Java/Scala development. XML template are directly embedded in the Scala generated code and type checked by the Scala type checker. Fig. 3 shows an excerpt of the Scala traits generated for the domain model meta-classes. It shows the use of immutable list and Option for 0..n reference and 0..1 reference respectively.

4.2 Memory footprint

To limit the dependencies, we decided to restrict the inheritance relationships in our generated code only to generated classes and to classes from the Java and the Scala frameworks. In this way dependencies are limited to the Java and Scala frameworks. A standalone JAR for the same metamodel in KMF has a size of 7 MB. After applying ProGuard, we obtain a JAR of 1.7MB. Indeed, the Scala dependencies load many packages that are not used by KMF, such as Scala-Swing, Scala-actors, etc.

[10] http://www.scala-lang.org/api/current/scala/Option.html

```
trait Transition extends FsmSampleContainer {
    private var input : java.lang.String = ""

    private var output : java.lang.String = ""

    private var source : fsmSample.State = _

    private var target : fsmSample.State = _

    private var action : Option[fsmSample.Action] = None
    ...
}


trait FSM extends FsmSampleContainer {

    // 0..n reference
    private var ownedState : scala.collection.mutable.ListBuffer[
        fsmSample.State] = scala.collection.mutable.ListBuffer[fsmSample
        .State]()
    ....
    // 0..n reference method helpers
    def getOwnedState : List[fsmSample.State] = {
        ownedState.toList
    }
    def getOwnedStateForJ : java.util.List[fsmSample.State] = {
        import scala.collection.JavaConversions._
        ownedState
    }

    def setOwnedState(ownedState : List[fsmSample.State] ) {
        this.ownedState.clear()
        this.ownedState.insertAll(0,ownedState)
        ownedState.foreach{e=>e.setEContainer(this,Some(()=>{this.
            removeOwnedState(e)}))}

    }

    def addOwnedState(ownedState : fsmSample.State) {
        ownedState.setEContainer(this,Some(()=>{this.removeOwnedState(
            ownedState)}))
        this.ownedState.append(ownedState)
    }

    def addAllOwnedState(ownedState : List[fsmSample.State]) {
        ownedState.foreach{ elem => addOwnedState(elem)}
    }

    def removeOwnedState(ownedState : fsmSample.State) {
        if(this.ownedState.size != 0 ) {
            this.ownedState.remove(this.ownedState.indexOf(ownedState))
            ownedState.setEContainer(null,None)
        }
    }

    def removeAllOwnedState() {
        this.ownedState.foreach{ elem => removeOwnedState(elem)}
    }

    def getClonelazy(subResult : java.util.IdentityHashMap[Object,Object
        ]): Unit = {
      ...
    }
}
```

Fig. 3. Excerpt of generated Scala code for domain meta-classes

Consequently, KMF has successfully been used on top of Dalvik [11], Avian [12], JamVM[13] or JavaSE for embedded Oracle Virtual Machine [14].

4.3 Multi-thread access

Model@Runtime serves as a common software reflection layer concurrently exploited by many processes; protection against such accesses may be coarse or fine grain.

At fine grain the model essentially needs to be read concurrently while allowing modifications. The KMF generated code realizes such protection by internally using mutable collections (for performance reasons) but only exposing cloned immutable list to outside via its public API. As a result processes can navigate the cloned list while others perform CRUD operations on it. Each process needs to actively ask a new cloned version to access to the modifications. Moreover, the mutator methods (setter) can be protected behind synchronized blocks.

At coarse grain the model representation is entirely hidden behind a safe model care tracker as in the Memento pattern [6]. This safe model care tracker systemically clones the model on *get* operations and keeps a master representation. This structure is particularly useful to keep an history of model representation at runtime. KMF can optionally generate such structure using Scala actors to protect concurrent access (get / put) to model care tracker.

4.4 Loader, Serializer and Cloner

When working with models, two tasks are essential and used before and after each action on a model, namely marshalling and unmarchalling.
Where EMF offers a generic loader we propose to use the generation phase to also generate a specific loader for each meta-model.
The EMF generic loader takes a model to load and its meta-model as parameters and intensively uses reflection mechanisms to perform the loading task. If this kind of mechanism allows creating a single loader, its usage is not efficient. The generated KMF code then provides meta-model specific loader, saver and cloner to improve their efficiency. We use the XML API which is part of the Scala standard library to parse and print XMI representations of object models, with no need for extra dependencies, and because it is efficient.
The loading and cloning are performed in two phases. The first phase consists in traversing the models for creating the objects in the order they are found in the XMI file. The last step links the objects together according to the references previously cached. Currently, the Maven plugin we propose is only able to generate

[11] http://www.dalvikvm.com/
[12] http://oss.readytalk.com/avian/
[13] http://jamvm.sourceforge.net/
[14] http://www.oracle.com/technetwork/java/embedded/downloads/javase/index.html

loaders and serializers for the XMI file format. EMF compatibility is obtained through the XMI file format. A direct API compatibility can be performed when EMF will separate the Ecore interfaces and Ecore implementation in different bundles to avoid useless dependencies.

4.5 Experiment and synthesis

	EMF	*KMF*	comparison
Model creation	376 ms	313 ms	1.2 times faster
Model clone	3588 ms	398 ms	9 times faster
Model save	7021 ms	2630 ms	2.66 times faster
Memory footprint	104MB of heap memory	61MB of heap memory	1.70 times lighter

Table 2. EMF and KMF efficiency

Besides, the memory footprints used to store, load, save or clone a model has decreased compared to the reference EMF implementation. To measure the memory footprints, (un)marshalling and cloning, we do the same experiment. We programmatically create models with 100 000 States with a transition between each state and an action in each Transaction.
The results for KMF are the following. On a Dell Precision E6400 with an Intel 2.5GHz iCore I7 CPU and 16GB of RAM, it takes 313ms to create the models, 2630 ms to save it in a file, 61 Mbytes of Heap memory, 398ms to clone and 3000s to load the model from a file[15]. Table 2 highlights the quantitative performance comparison results between EMF and KMF.

Table 3 provides the qualitative comparison results between EMF and KMF.

	EMF	*KMF*
Memory footprints	$\times$(104MB)	$\vee$(61MB)
Dependencies	$\times$ (15MB)	$\vee$ (Scala standard library) (1.7MB)
Lazy Loading	$\vee$	$\vee$ (Proxy support)
Thread safety	$\times$	$\vee$ (immutable lists, no registry)
Efficient model (un)marshalling and cloning	$\sim$	$\vee$ (see Table 2)
Design tools compatibility	$\vee$	$\vee$ (through XMI compatibility)

Table 3. EMF and KMF regarding models@runtime requirements

[15] This experiment can be download http://goo.gl/9Huwa (for eclipse project) or http://goo.gl/0sWRo (for the maven project)

5 Discussion

5.1 Refactoring impact

The first major consequence of removing the runtime dependencies with EMF is that all the methods defined in `EObject` are now unavailable. This could have a significant impact on the existing code that uses these methods. In order to limit the refactoring impact of this removal, we re-implemented the `eContainer` mechanism of EMF in our generated code.

Another important design choice we made for KMF was to use Scala as the default language for the generation and use of the code. Java has also been considered as a language since Scala code is fully compatible with Java. However, Scala code is not always friendly to use from a Java program[16]. To ease the use of KMF in a Java environment, we also provide a standard Java API, which in particular exposes Java lists, by duplicating some methods that are suffixed with "4J". That requires all model navigation-related code to be rewrote to use these new methods. The dual strategy could of course have been implemented: generating Java code and exposing in addition a Scala API.

The main rationale behind the choice of Scala was that the Scala standard library provides many facilities that are useful for Models(@runtime). In particular, the systematic introduction of Scala `Option` for each optional (*i.e.* having lower bounds equals to 0) attribute or reference implies to explicitly test if the element is defined or not, in a neater way than `if (myRef == null)` in Java, and in a safer way than a `NullPointerException` popping at runtime. Here again, this mechanism enforces developers to consider the optional aspect of these elements and avoids lots of null-checks, but requires a deep refactoring.

5.2 Limitations

KMF has formerly been developed and tested using the Kevoree metamodel originally designed with EMF tools. This first step allowed for setting up the basis for model, loader and serializer generation. The use of a fairly different metamodel (from Kermeta [12]) highlighted some missing features in the generation process, and strongly helped in improving KMF. However, KMF still has some limitations.
For instance, reverse relations have already be flagged as missing in the generated code.

EMF allows model elements to have some relations with elements from other metamodels (references, attribute types, inheritance, etc). This mechanism has been partially realized on a concrete use case, but it still need improvements and implementation discussions.

[16] To give an idea, a Scala list built by concatenating the empty list (Nil) and the element 1 would be written `1::nil` in Scala. The construction of the same Scala list in Java would yield `$colon$colon$.MODULE$.apply((Integer) 1, nil);`

Moreover, the generation of loaders and serializers relies on containment relations between model elements, and it requires a root container element. Until now, we considered metamodels that have only one single model element as root of the containment tree, but this is not the general case. Indeed, all model elements must have a container, but not necessarily under a single root for the metamodel. There could be several containment roots (and several containment trees) in a metamodel, with references to each other, but also across different metamodels. Generators of loaders and serializers are not ready to accept such kind of metamodels, but meeting this requirement is already considered as future work.

KMF addresses performances issues of models in memory (heap). Complementary approaches like CDO addresses the management of models in persistence memory (databases). The CDO (Connected Data Objects) Model Repository [17] is a distributed shared model framework for EMF models and metamodels. CDO has a 3-tier architecture supporting EMF-based client applications, featuring a central model repository server and leveraging different types of pluggable data storage back-ends like relational databases, object databases and file systems. The default client/server communication protocol is implemented with the Net4j Signalling Platform. This solution offers tools to easily do collaborative work and history management. However, it uses EMF as a local representation. Consequently it inherits a part of the drawbacks coming from EMF (e.g. dependencies for client side, memory footprints). Moreover the use of external server to manage history is not useful for pervasive systems that may have a sporadic network and even if it can embed server side on client, the overhead of the dependencies is not suitable for lightweight systems.

6 Conclusion

This position paper has discussed the needs for adapting the *de facto* standard in the MDE community, *i.e.* the Eclipse Modelling Framework (EMF), for a more dynamic usage of models in the context of Models@Runtime. After highlighting requirements related to Models@Runtime, this paper has presented an initial adaptation of EMF, named Kevoree Modelling Framework (KMF), implemented in Scala and generating code for this language. Even if KMF only supports XMI serialisation, it provides a significant speedup on model creation, model (un)marshalling and model cloning. It also has a lighter memory footprint than the reference implementation, and its runtime dependencies are limited to the Java and Scala libraries, whereas the EMF generated code has tight dependencies to Eclipse and Equinox. This significantly hinders the reusability of the EMF code outside Eclipse, while KMF code can run Eclipse-free on various Java virtual machines. Finally, and unlike EMF which is not thread-safe, KMF provides a built-in support for in-memory safe concurrent access to models.

[17] http://www.eclipse.org/cdo/

KMF is still at an early stage of existence and needs to be improved through usage. Future work on KMF already addresses the limitations and points discussed in section 5. Independently from the improvement of existing features of KMF, we think that additional tools could promote its adoption.

Set operations. Model merging or model comparison are common operations implemented by tools that use models as representations of their internal data. Implementing mergers or comparators is often a complex, lengthy and error prone task. As a future work, we plan to offer the possibility to generate metamodel specific set operations such as union, difference or intersection. These operations could decrease the complexity of implementing model mergers.

Customizable generation plugin. In its current implementation, the plugin allows for generation of all features (model, cloner, loader and serializer) or only model and cloner. This customization of the plugin behavior will be improved to enable the separate generation of each feature. Moreover, the loader and serializer generators are hard coded in the plugin. It is thus problematic to use other generators to create loaders and serializers that use another serialization format (namely XMI). In the future, we plan to improve the plugin parameterization to allow users to change the generators. This will also enable the seamless integration of other generators (e.g. to create set operations).

References

1. Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Mocas: A state-based component model for self-adaptation. In *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009, San Francisco, California, USA, September 14-18, 2009*, pages 206–215. IEEE Computer Society, 2009.
2. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
3. Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. 5525:1–26, 2009.
4. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.
5. François Fouquet, Olivier Barais, and Jean-Marc Jézéquel. Building a kermeta compiler using scala: an experience report. In *Workshop Scala Days 2010*, Lausanne, Switzerland, 2010.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
7. John C. Georgas, André van der Hoek, and Richard N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10):52–60, October 2009.

8. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin / Heidelberg, 2009.
9. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
10. Bernhard Merkle. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 139–148, New York, NY, USA, 2010. ACM.
11. Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.
12. P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.
13. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
14. Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.
15. Fredrik Seehusen and Ketil Stlen. An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin / Heidelberg, 2011.
16. Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.