Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Evaluation of Kermeta for Solving Graph-based Problems

Naouel Moha, Sagar Sen, Cyril Faucher, Olivier Barais, Jean-Marc Jézéquel

IRISA / INRIA Rennes Bretagne Atlantique, e-mail: {moha, ssen, cfaucher, barais, jezequel}irisa.fr

Received: date / Revised version: date

Abstract. Kermeta is a meta-language for specifying the structure and behavior of graphs of interconnected objects called models. In this paper, we show that Kermeta is suitable for solving graph-based problems. First, Kermeta allows the specification of generic model transformations such as refactorings that we apply to different metamodels including Ecore, Java, and Uml. Second, we demonstrate the extensibility of Kermeta to the formal language Alloy using an inter-language model transformation. Kermeta uses Alloy to generate recommendations for completing partially specified models. Third, using a common case study we show that the Kermeta compiler achieves better execution time and memory performance compared to similar graph-based approaches. The three solutions proposed for graph-based problems correspond to the first contribution of the paper. The second contribution is the comparison of these solutions with those proposed by other graph-based tools. The evaluation of Kermeta according to the criteria of genericity, extensibility, and performance represents the third contribution.

Key words: MDE – Metamodelling – Model Typing – Model Transformation – Refactoring – Performance – Genericity – Extensibility – Alloy

1 Introduction

Model-Driven Engineering (MDE) is a software development methodology which focuses on models as first-class entities. Models are graphs of objects interconnected by bidirectional relationships defined in metamodels. MDE aims to improve productivity of developers by maximizing compatibility between systems and platforms and simplifying the process of design.

Kermeta has been developed as a core language for an MDE platform. It is an executable metamodelling language implemented on top of the Eclipse Modeling Framework (EMF) within the Eclipse development environment.

In this paper, we present Kermeta as a suitable language for solving graph-based problems. We focus on three case studies proposed in the GraBats'08 tool contest[1] that involve specific graph-based problems.

The first case study consists of applying three well known refactorings [6] (Encapsulate Field, Move Method, and Pull-up Method) on models of Java programs. In this paper, we present a generalised approach to model refactoring that is applicable not only to Java programs but other metamodels such as Ecore and Uml. We specify the generic refactorings for various metamodels using the notion of model typing [28], which is an extension of object typing in the model-oriented context.

The second case study (conference scheduling) highlights the extensibility of Kermeta to external languages, such as the formal language Alloy. We present an inter-language model transformation from Kermeta to Alloy to complete partial models. In the context of the case study, we use Alloy to generate different valid schedules for an unscheduled conference.

The third case study is an AntWorld simulation demonstrating Kermeta performance with regard to execution time and memory usage. The Kermeta to Java / EMF compiler provides a version of the Kermeta simulation achieving better performance compared to similar approaches based on Eclipse and eventually EMF.

Our contributions are threefold: (1) approaches to solve graph-based problems involving these case studies, (2) experiments to evaluate Kermeta in terms of genericity, extensibility, and performance using these case studies, (3) and a comparison of our approaches with those proposed by other graph-based tools.

This article is organized as follows. Section 2 introduces Kermeta and highlights some of its features in-

[1] The GraBats'08 tool contest was held during the 4^{th} International Workshop on Graph-Based Tools 2008.

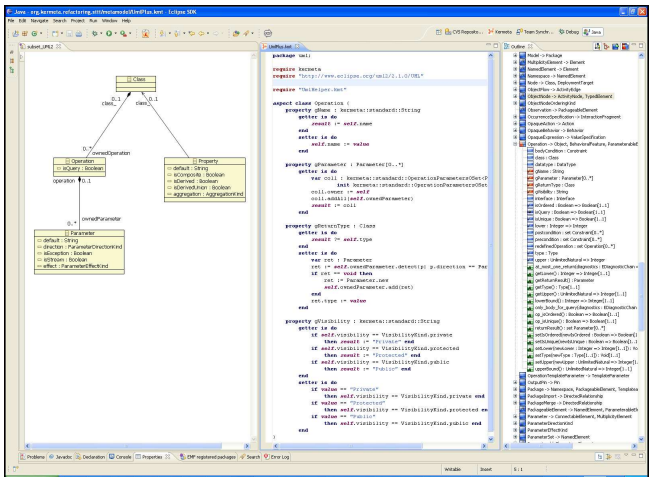

Fig. 1. Kermeta Graphical Interface

cluding the notion of model typing. Sections 3, 4, and 5 develop each of the three criteria that characterise Kermeta based on the three case studies. Section 6 surveys related work. Section 7 concludes and presents future work.

2 Kermeta

2.1 Description

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [22]. The object-oriented meta-language MOF supports the definition of metamodels in terms of object-oriented structures (packages, classes, properties, and operations). It also provides model-specific constructions such as containments and associations between classes. Kermeta extends the MOF with an imperative action language for specifying constraints and operational semantics for metamodels [20]. Kermeta is built on top of EMF within the Eclipse development environment. The action language of Kermeta provides mechanisms for dynamic binding, reflection, and exception handling. It also includes classical control structures such as blocks, conditionals, and loops. Figure 1 shows the Kermeta graphical interface within Eclipse. It includes three views: a graphical representation of the current metamodel, a Kermeta code that applies transformations on the metamodel, and a tree representation of concepts of one of the two first views depending on the current view.

In the next paragraphs, we describe two key features of the Kermeta language essential for the comprehension of the paper: its ability of extension and the notion of model typing.

2.2 Extension of Kermeta

The first key feature of Kermeta is its ability to extend an existing metamodel with constraints, new structural elements (meta-classes, classes, properties, and operations), and functionalities defined with other languages. This feature offers more flexibility to developers by enabling them to easily manipulate and reuse existing metamodels. The static composition operator "*require*" allows defining these various aspects in separate units and integrating them automatically into the metamodel. The composition is performed statically and the composed metamodel is type-checked to ensure the safe integration of all units. This mechanism can be compared to the open class paradigm [5]. Open classes in Kermeta are used to organize "cross-cutting" concerns separately from their metamodel, a key feature of aspect-oriented programming [12]. Thanks to this composition operator, Kermeta remains a kernel platform and safely integrate all concerns around a metamodel.

Kermeta offers expressions very similar to Object Constraint Language (OCL) expressions [24]. In particular, Kermeta includes lexical closures similar to OCL iterators on collections such as each, collect, select, or detect. Moreover, Kermeta also allows the direct importation and evaluation of OCL constraints. Pre-conditions and post-conditions can be defined for operations and invariants on classes.

Kermeta and its framework remain dedicated to model processing but provide an easy integration with other languages. Kermeta also allows importing Java classes to use services such as file input/output or network communications, which are not available in the Kermeta framework. It is also very useful, for instance, to make models communicate with existing Java applications. An other example of integration is the extension to Alloy [9] as presented in Section 4.

2.3 Model Typing

The second key feature of Kermeta is the notion of model typing [28]. It consists of a simple extension to object-oriented typing in a model-oriented context. A model typing is a strategy for typing models as collections of interconnected objects. Model typing permits the detection of type errors early in the design process of model transformation. Moreover, it allows more flexible reuse of model transformations across various metamodels, while preserving type safety [28]. Type safety is guaranteed by type conformance, a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type groups and type group matching [3]. The matching relation, denoted <#, between two model types defines a function of the set of object types they contain according to the following definition adapted from [28]:

> Model type *M'* matches another *M* (denoted *M'* <# *M*) iff for each object type *C* in *M*, there is one and only one corresponding object type *C'* in *M'* such that every property and operation in *M.C* also occurs in *M'.C'* with exactly the same signature as in *M.C*.

A limitation of the model typing is the name-dependent structural conformance. Indeed, only the matching of variants of model types that have respective objects interconnected in the same structural manner and with identical properties and operations names is possible. To reduce this limitation, we added two mechanisms to the model typing: the renaming of properties/operations and the possibility to match with other objects belonging to the same inheritance hierarchy. The renaming consists in specifying a new name for a property or an operation in a specific metamodel to allow the matching. The second mechanism consists in extending the matching to parents and children of the current object if this one does not match.

The following sections describe the three criteria of Kermeta using a common pattern. First, we describe the problem that we propose to solve. Second, we illustrate our approach using a case study. Third, we present some experiments and results. Finally, we provide a comparison with other graph-based tools[2] followed by a discussion.

3 Genericity

In this section, we present generic model transformations in Kermeta based on the notion of model typing. More precisely, we specify generic refactorings for different metamodels.

3.1 Problem Description

Refactoring has been intensively investigated in the graph and model transformation community over the last few years [16]. However, to the best of our knowledge, there exists no approach to specify metamodel independent generic transformations. In current approaches, the specification of refactorings are highly dependent on the metamodel. Our goal is to specify generic model transformations, such as refactorings, that can be reusable on different metamodels. For instance, a refactoring such as Pull-up Method (*i.e.*, moving methods to the superclass if these methods have same signatures on subclasses) could be generic across any language supporting the object-oriented notion of inheritance (UML, ECORE, Java).

3.2 Case Study: Refactoring

The refactoring case study of the GRABATS'08 tool contest [8] consists of applying three well known refactorings (Encapsulate Field, Move Method, and Pull-up Method) on models of Java programs. We generalise this case study to the problem of specifying generic refactorings for various metamodels. We consider not only models of Java programs but also ECORE and UML models.

[2] These tools participate to the tool contest organised for the STTT Special Section on Graph-based Tool Comparison.

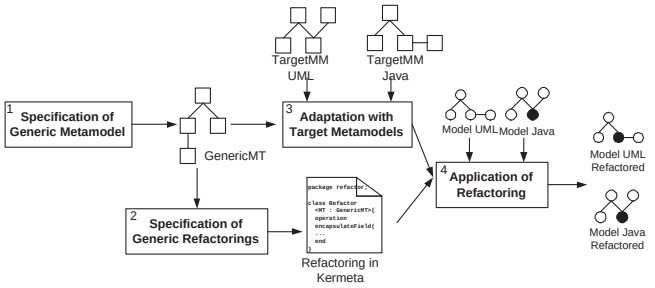

Fig. 2. Approach for the Specification of Generic Refactorings

We illustrate our approach for the Encapsulate Field refactoring. We recall that the Encapsulate Field refactoring consists of making a public field private and providing accessors [6].

3.3 Approach: Specification of Generic Refactorings

Figure 2 illustrates the four steps of our approach for the specification of generic refactorings. The first step consists in specifying a generic metamodel `GenericMT`[3], which is a super-type of all metamodels. Then, in the second step, we specify a refactoring in Kermeta using `GenericMT` as the source metamodel. In the third step, the target metamodels such as UML or Java are adapted to match with the metamodel `GenericMT`. The target metamodels are then subtypes of `GenericMT`. In the last step, the refactoring can then be concretely applied to all models of all target metamodels.

3.3.1 Step 1: Specification of Generic Metamodel

Our approach consists first of specifying a lightweight metamodel that contains the minimum required classes for specifying refactorings. The generic metamodel, called `GenericMT`, given in Figure 3 has been designed to specify refactorings. `GenericMT` consists of concepts such as classes, properties, operations, and parameters common to all metamodels. We use the letter 'g' as a prefix in the name of each element of the metamodel to denote the fact that they actually play the role of formal generic parameters. The elements contained within the metamodel `GenericMT` are minimum and sufficient for the specification of the three refactorings.

3.3.2 Step 2: Specification of Generic Refactorings

In the second step, we specify refactorings based on the generic metamodel. Listing 1 gives a Kermeta code excerpt of the refactoring Encapsulate Field based on the metamodel `GenericMT`. This code checks (using preconditions) if for the specified `field` a `getter` accessor does not exists or if it exists, it is static. In the body of the operation `encapsulateField`, it creates a getter accessor if it did not exist yet. This code gives an insight into the specification of refactorings in Kermeta. The interested reader can refer to the Kermeta syntax in [11].

[3] MT refers to Model Type

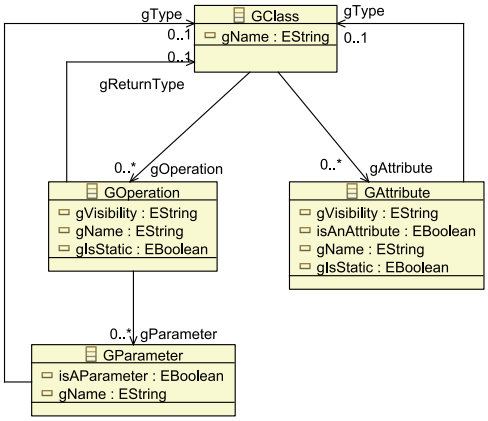

Fig. 3. Generic Metamodel `GenericMT`

```
package refactor;

class Refactor<MT : GenericMT> {

    operation encapsulateField(
                field : MT::GAttribute,
                fieldClass : MT::GClass,
                getterName : String,
                setterName : String) : Void

    // Preconditions
    pre getterIsStatic_NotExist is do
            if fieldClass.gOperation.exists{op |
                    op.gName == getterName}
            then
                fieldClass.gOperation.detect{op |
                    op.gName == getterName}.gIsStatic
                        == field.gIsStatic
            else
                true
            end
    end

    // Method body
    is do

        // Manage the getter
        if not fieldClass.gOperation.exists{ op |
            op.gName == getterName
        } then
            // No getter, so we must add it
            var op : MT::GOperation init MT::GOperation.new
            op.gName := getterName
            fieldClass.gOperation.add(op)

            op.gReturnType := field.gType
        end

        // Manage the setter and the visibility
        ...
    end
}
```

Listing 1. Kermeta Code for the Encapsulate Field Refactoring.

3.3.3 Step 3: Adaptation with Target Metamodels

The third step consists of adapting the target metamodels to the generic metamodel `GenericMT` using the mechanism of aspects in Kermeta. The adaptation consists in weaving in the target metamodel properties and operations that match with those of the generic metamodel. For each operation, we specify the body of the method. For each property, we specify how to set and get it by defining getter and setter accessors. This step is necessary because the model typing is not sufficient for matching metamodels that are structurally too different. Thus, the adaptation modifies the structure of the target metamodel with additional elements, and uses the model typing to match metamodels.

During the matching process, we match with the target metamodel not only one class of the generic metamodel but a set of classes. Thus, the model type conformance is hard to obtain if it is not possible to distinguish classes of the generic metamodel that have same name attributes. We introduced a "non-matching" strategy by adding a discriminant attribute such as `isAnAttribute` in `GAttribute` to distinguish this class from `GOperation`.

Listings 2 and 3 show the adaptation of the return type of an operation for the Java and UML target metamodels. The adaptation for Java is quite straightforward and consists of assignments. The adaptation for UML requires in the setter to look for a parameter with `ParameterDirectionKind.return` as a value of the attribute `direction`.

```
package javaprogram;

require "platform:/resource/org.kermeta.refactoring/
     metamodel/JavaProgram.ecore"

aspect class Operation {

    property gReturnType : javaprogram::Class
        getter is do
            result ?= self.type
        end
        setter is do
            self.type := value
        end
}
```

Listing 2. Kermeta Code for Java Adaptation.

```
package uml;

require "http://www.eclipse.org/uml2/2.1.0/UML"

aspect class Operation {

    property gReturnType : uml::Class
        getter is do
            result ?= self.type
        end
        setter is do
            var ret : uml::Parameter
            ret := self.ownedParameter.detect{p|
            p.direction == ParameterDirectionKind.return}

            if ret == void then
                ret := Parameter.new
                self.ownedParameter.add(ret)
            end
            ret.type := value
        end
}
```

Listing 3. Kermeta Code for UML Adaptation.

3.3.4 Step 4: Application of Refactoring

The last step of our approach consists of applying the refactoring on the target metamodel as illustrated in Listing 3 for the UML metamodel. We can notice that the class `Refactor` takes as argument the metamodel UML, which thanks to the adaptation of Listing 3 is a subtype of the expected super-type `GenericMT` as specified in Listing 1. The models to refactor are loaded and saved after refactoring in XMI files.

```
package refactor;

require "http://www.eclipse.org/uml2/2.1.0/UML"

class Main {
    operation main() : Void is do

        var refactor : refactor::Refactor<uml::UmlMT> init
             refactor::Refactor<uml::UmlMT>.new

        var nameField  : uml::Property
        var fieldClass : uml::Class

        refactor.encapsulateField(nameField, fieldClass,
                                  "getName", "setName")
    end
}
```

Listing 4. Kermeta Code for Applying the Encapsulate Field Refactoring on the UML metamodel.

3.4 Experiments and Results

We specify and apply three refactorings suggested in the GraBats'08 tool contest (Encapsulate Field, Move Method, and Pull-up Method) on three different metamodels (Uml, Ecore, and Java Program). The metamodels structurally differ. For example, the Uml metamodel has a hierarchical structure whereas the Java metamodel has a flat structure (*i.e.*, has no containers).

3.5 Comparison with Other Tools

None of the tools that participate to the contest offer the possibility to specify generic refactorings. However, they focus on other criteria. Tools such as Fujaba [7], Progres [32], and VMTS [18] focus on the user interaction criterion. For example, VMTS provides a source code like presentation and control flow diagrams for rewriting rules. In Kermeta, it is also possible to visualize models and metamodels using the Ecore Diagram Editor available in the Eclipse Ecore Tools plugin or graphical editors generated with GMF (Graphical Modeling Framework) plugin. Jdt2Mdr [19], Progres [32], and Fujaba [7] focus on the expressiveness and extensibility criteria. Jdt2Mdr transforms Uml models of controlled graph transformations into executable Java code. Progres provides imperative control structures such as conditional branches and iterations. Such control structures enhance expressiveness in specifying graph transformation rules.

3.6 Discussion

Writing adaptations can be more or less difficult depending on the developers' knowledge of the target metamodel. However, after adaptation, the developers can reuse all model transformations written for the generic metamodel. Conversely, if we write a transformation for the generic metamodel, we can apply it on all target metamodels.

Our approach supports model evolution. Indeed, if a metamodel evolves but still matches with the generic metamodel, the transformations are still valid for all models of the new metamodel.

Our approach contributes to the development of generic tools independent of the metamodelling language. In future work, we intend to investigate generic analyses to compute metrics and detect patterns and anti-patterns or inconsistencies in different metamodels.

4 Extensibility

Model transformations can extend Kermeta to external languages and model-oriented tools with functionality not readily available in Kermeta. In the following subsections, we demonstrate extensibility of Kermeta by presenting a transformation from metamodels in Kermeta to declarative specifications in the formal language Alloy [9].

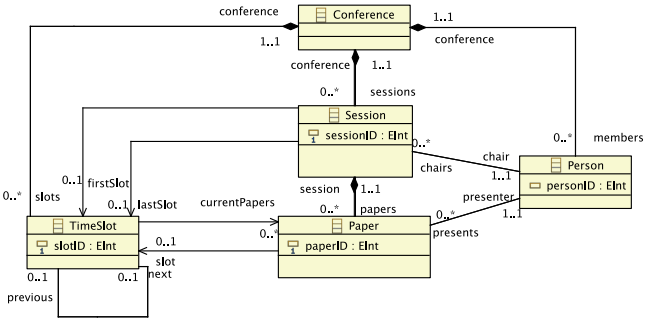

Fig. 4. Conference Scheduling Metamodel MM_{cs}

4.1 Problem Description

We define a *partial model* as *graph of objects* that is either inconsistent with the structure of its metamodel or does not satisfy some invariants on its metamodel. The problem we address is the automatic transformation of a partial model into a *complete model*. The complete model is a modification of the partial model such that it becomes consistent with its metamodel and invariants.

We can automate the process of completing a partial model using a *constraint satisfaction language* (CSL) equipped with a *solver*. We name this process *automatic model completion*. We want to extend Kermeta with this capability. The extension transformation must bridge the gap in expressiveness between Kermeta metamodels + invariants and the target CSL. This is an issue because most CSLs have concise grammars and well-defined declarative semantics as opposed to domain-specific Kermeta metamodels. Further, we must transform low-level solutions *back* from constraint solvers to domain-specific complete models conforming to a Kermeta metamodel.

4.2 Case Study: Conference Scheduling

This case study describes a modelling domain for scheduling papers in different sessions of a conference. We structure the concepts of the conference scheduling problem domain in the metamodel MM_{cs} shown in Figure 4. The metamodel MM_{cs} consists of a conference with sessions. Each session contains papers for presentation and a session chair. Every paper has a presenter. All papers in the conference must be assigned to time slots while respecting invariants I_{cs} such as:

1. No simultaneous papers are presented by the same person;
2. No presenter is chairing another session at the same time;
3. Nobody chairs two sessions simultaneously.

A partial model containing objects of classes in MM_{cs} is shown in Figure 5. In the partial model, we *do not assign papers to time slots* rendering the invariants I_{cs} unsatisfied.

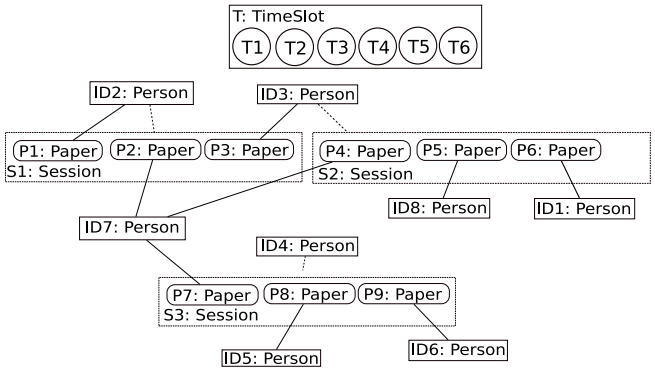

Fig. 5. A Partial Model p_{cs} Without Slot Assignment

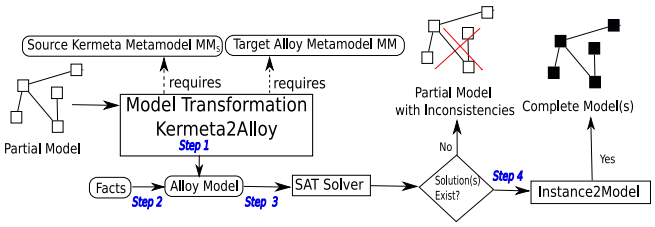

Fig. 6. Extensibility Transformation: Kermeta to Alloy for Partial Model Completion

4.3 Approach: Partial Model Completion

We outline our approach for automatic model completion in Figure 6. Our approach consists of the four following steps:

Step 1: The model transformation Kermeta2Alloy: (a) Transforms a source Kermeta metamodel MM_s to an ALLOY model A_s. It transforms MM_s classes, their properties and *implicit constraints* (inheritance and property characteristics such as opposites, composition, uniqueness, and identity) to ALLOY signatures and facts. *Example:* Listing 5 shows the ALLOY signature corresponding to the transformation of the class Paper in MM_{cs} and the facts generated from implicit constraints in MM_{cs}.

```
sig Paper
{ paperID : lone Int,
  session : one Session,
  presenter : one Person,
  slot : one TimeSlot
}

fact Paper_containers
{ all o: Paper | o in Session.papers }

fact Session_papers_Paper_session_opposite
{ all  o1 : Session, o2 : Paper |o2 in o1.papers implies o1
     in o2.session }
```

Listing 5. Alloy Model for Conference Scheduling

(b) Transforms a source partial model p_s to ALLOY predicates and appends them to A_s. *Example:* The partial model in Figure 5 is transformed to the predicate $ConferenceModel$, partially presented in Listing 6. The predicate states the number of objects in the partial model. Following this, it states values for properties that we extract from the partial model. We assign values for all properties available in the partial model including sessions, papers, presenters, and time slots. What remains unassigned in the predicate are the properties for Paper.slot.

```
pred ConferenceModel
{
/*Number of objects of each signature in Partial Model*/
#Conference=1 and #Session=3 and #Paper=9 and
#Person=8 and #TimeSlot=6 and
/*Exists some object of a signature with properties of
     values in the partial model*/
some s1: Session, s2:Session ,s3:Session |
s1.sessionID = 10 and s2.sessionID= 20 and s3.sessionID
     =30 and
...
/*Similar expressions that define other objects in
     partial model */
}
```

Listing 6. Alloy Predicate for p_{cs}

(c) Inserts a run command to solve p_s in A_s. The run command states the scopes for the different objects we expect in the complete model and for integers between -2^7 to 2^7. It obtains these scopes from the partial model. *Example:* In the case study we use integers to specify identities for sessions, papers, and people. We create the ALLOY run command in Listing 7 and insert it into A_{cs}. The run command solves the predicate ConferenceModel.

```
run ConferenceModel  for 1 Conference,6 TimeSlot, 9 Paper,
     3 Session, 8 Person, 7 int
```

Listing 7. Alloy Run Command to Complete Partial Model

Step 2: We insert ALLOY facts for invariants I_s to the metamodel MM_s. These invariants are initially specified in OCL. At the moment, we manually transform natural language or OCL constraints to ALLOY facts representing these invariants. In future, we intend to integrate an automatic transformation of a subset of OCL to ALLOY facts into Kermeta2Alloy. *Example:* In the conference scheduling case study, one of the invariants states that *a person cannot give simultaneous presentations*. We encode both the OCL version and its ALLOY fact in Listing 8.

```
//No Simulatenous Paper Presentations by a Person
//OCL Version
context Person
  inv noSimultaneousPresentations:
    self.presents.forAll{ p1 , p2 | p1!=p2 implies p1.slot
         !=p2.slot}
//Transformed Alloy Fact
fact noSimultaneousPresentations
{ all p:Person| all paper1 : p.presents, paper2: p.presents
     | paper1!=paper2 implies paper1.slot!=paper2.slot }
```

Listing 8. OCL Invariant to Alloy Fact

Step 3: We invoke the ALLOY API to transform the ALLOY source model A_s to boolean conjunction normal form (CNF). We solve the CNF formula using an off-the-shelf satisfiability (SAT) solver such as ZChaff [14] or MiniSAT [21] to obtain solution(s) (if they exist). These solutions are dumped as ALLOY XML files. *Example:* We present the results of executing the run statement of Listing 7 in Section 4.4.

Step 4: We transform an ALLOY XML file representing a solution to a complete model c_s using the transformation Alloy2Model. This complete model c_s conforms to the metamodel MM_s. *Example:* In this transformation low-level relational mappings between atoms is transformed to a conference model conforming to MM_{cs}.

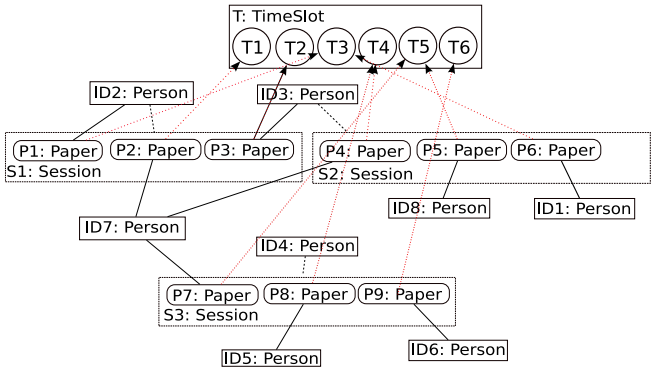

Fig. 7. A Complete Model c_{cs}

4.4 Experiments and Results

We name the ALLOY model we obtain for conference scheduling A_{cs}. We solve A_{cs} to obtain a complete model c_{cs} with a valid conference schedule. We execute a run command to solve the predicate ConferenceModel in A_{cs}. The result of our experiment is an XMI file or a set of XMI files or complete conference model(s) with valid schedule(s). One of the complete models c_{cs} in comprehensible visual syntax is shown in Figure 7. The red dotted arrows from papers to slots illustrate the scheduling solution.

4.5 Comparison with other Tools

The AGG-EMT [29] tool performs scheduling using triple graph grammars (TGG). In their approach, a paper is assigned a time slot using TGG rules such that negative application conditions (NACs) are not violated. The termination of the transformation process determines if we can schedule the conference. The AGG solution is specific to the scheduling problem while our approach is generic and applicable to any Kermeta metamodel transformable to ALLOY. However, the domain specific TGG rules can make the scheduling process itself more efficient than SAT solving.

4.6 Discussion

Our approach is generic and valid for any metamodel and not just for the conference scheduling case study. We present this approach in the context of automatic model completion but we have applied this approach earlier for model completion in model editors [27], and for test model generation [26]. There has been other contributions to transform high-level languages to ALLOY such as the prototype tool UML2Alloy [1].

Kermeta supports import/export to industry standards for metamodel and metadata specification such as EMF and XSD allowing widespread application of transformations written in Kermeta. For instance, the transformation to ALLOY for any metamodel makes them amenable to formal analysis, automatic model synthesis, and counter-example generation.

5 Performance

In this section, we implement in Kermeta the AntWorld simulation and evaluate Kermeta performance with regard to execution time and memory usage. We also compare and discuss these performance results to those obtained with similar approaches based on ECLIPSE and eventually EMF.

5.1 Problem Description

Performance and scalability issues are often seen as a challenge to promote model-driven tools to industry. Indeed, although model-driven applications might offer high-level design abstractions and reduce development time and efforts, they might also experience scalability problem with respect to performance.

5.2 Case Study: AntWorld

The AntWorld simulation is a case study designed as a benchmark for the comparison of graph-based tools [33]. It aims to run tools on a scalable application to evaluate their performance in terms of execution time and memory usage.

The case study simulates an ant colony searching for food around the area of the ant hill located in the center of a grid. If an ant finds food, it drops pheromones in its way back home. An ant that brings food into the ant hill leads to the creation of new ants. If an ant searching for food hits a pheromone, the ant follows the pheromone path to the food. The simulation is divided in round and every ant moves during a round.

Figure 8 gives the ECORE metamodel of the AntWorld simulation. The class `Ant` represents an ant and contains an attribute `mode` to determine whether the ant is searching for food or moving back to the center. The class `Map` represents the grid of nodes corresponding to the area of food search. A map contains a set of nodes represented by the class `GridNode`. Among the grid nodes, there are two special types of nodes `AxisNode` and `CenterNode`. The `AxisNode` is a node on an axis. The `CenterNode` corresponds to the node in the center of the grid. `CenterNode` inherits from `AxisNode` because it is at the intersection of the two axes. A `GridNode` is located in a given level and might be at the border of the map. It also contains a number of food parts and a number of pheromones.

5.3 Approach

Until recently, Kermeta applications were executed only in an interpreted mode. To get better performance, we have developed a Kermeta to Java/EMF compiler that allows developers to deploy Kermeta applications as Java / EMF and thus, execute Kermeta applications in a compiled mode. The Kermeta compiler first transforms a Kermeta model into an ECORE model and then generates an ECLIPSE plugin in Java/EMF source code. The

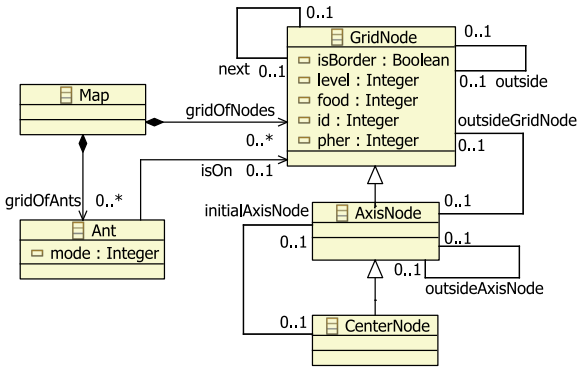

Fig. 8. AntWorld Metamodel

resulting Java source code may be used both in Eclipse application or in Java application (standalone). The Java compiled version of an Kermeta application typically runs faster than in interpreted mode.

5.3.1 Implementation

We implement the AntWorld simulation in Kermeta and in particular, the local search. The transformation rules of the AntWorld simulation are written as operations, which are sets of expressions that specify the expected behavior. The operations are added to the Ecore metamodel of AntWorld by using the aspect-oriented modeling facilities of Kermeta. Listing 9 describes the operation `antEat` weaved into the class `GridNode`. This operation describes the behavior of an ant that arrives at a node containing food. The ant takes a piece of food and drops 1024 parts of pheromones. This listing shows also a reference weaved into the class `Map` to add a cache of nodes with pheromones for improving the performance during the update of the number of pheromones on a node.

```
package antworld;

require kermeta
require "platform:/resource/AntWorld/AntWorld.ecore"

using antworld

aspect class Map {

   reference gridOfPhers : bag GridNode[0..*]

}

aspect class GridNode {

  operation antEat(map : Map) is do
        self.food := self.food - 1
        self.pher := self.pher + 1024

        // The current node is added to the cache
        // when it is initialized, i.e. self.pher == 1024
        if self.pher == 1024 then
            map.gridOfPhers.add(self)
        end
    end
}
```

Listing 9. Implementation by Aspect of the Reference `gridOfPhers` on the Class `Map` and the Operation `antEat` on the Class `Ant`.

5.4 Experiments and Results

We computed the execution time and memory usage of the AntWorld simulation written in Kermeta and executed in a compiled mode. These experiments were performed on a laptop with an Intel Core 2 CPU T2600, 2.16GHz, and 2Go for RAM under Windows XP. We also computed these performance tests using the version of Kermeta on the virtual desktop infrastructure (VDI) submitted for the tool contest to have a fair comparison with other VDIs submitted for the tool contest.

The table 1 presents the performance results for every 25 rounds. It gives the number of grid levels, ants, and nodes with pheromone. It also provides the elapsed time between every 25 rounds, the cumulative elapsed time, and the consuming memory. The elapsed time are given both on local and the VDI.

The memory usage of the Kermeta runtime context is constant during the execution and independent of the algorithm. The memory usage excludes the memory consumed for the runtime context (estimated at 5246 kB). It increases progressively with the number of rounds.

5.5 Comparison with other Tools

We compare the performance results of Kermeta with other tools partipating in the contest. The VMTS tool [17] is the fastest tool among all tools participating to the tool contest because it seems to be hard coded using C++. VMTS is 53.7 times faster than Kermeta at 500 rounds. We compare Kermeta with other tools based on Eclipse or EMF like Viatra2 [31] and EMF Transformation [2]. We execute the simulation using the VDIs to have a fair comparison. Figure 9 illustrates the results obtained with Viatra2 and Kermeta. We observe that Kermeta is 4.9 times faster than Viatra2 at 500 rounds. As regards EMF Transformation, results given during the GraBats'08 Workshop shows that Kermeta is 65 times faster at 100 rounds. In terms of memory usage, Kermeta seems to be the best solution with 10,353 kB at 500 rounds compared to VMTS with 31,404 kB (that is, a factor of 3) and 90,156 kB for GrGen [4] (a factor of 8.7). Viatra2 consumes 145,000 kB at 350 rounds (results directly provided by Viatra2) in comparison with 4,601 kB in Kermeta (a factor of 31,5).

5.6 Discussion

Viatra2 is not compiled, but interpreted. The use of a compiled version is very significant in Kermeta and especially in the case of the AntWorld simulation. The compiled version is 50 times faster than the interpreted one.

The measures could have been influenced by external elements. Indeed, we notice that running programs or network connections can reduce from 10% to 20% the time executions. Therefore, we compute our performance measures by disconnecting all network connections and closing all programs except Kermeta. However, parasite programs may still have influenced the measures.

Number of Rounds	Number of Grid Levels	Number of Ants	Number of Nodes with Pheromone	Elapsed Time (ms) (Local)	(VDI)	Cumulative Time (ms) (Local)	(VDI)	Memory (kB)
25	8	20	19	47	(157)	47	(157)	30
50	8	182	21	125	(226)	172	(383)	40
75	14	693	65	515	(841)	687	(1224)	131
100	18	1775	142	1313	(1785)	2000	(3009)	254
125	22	3070	277	2437	(3283)	4437	(6292)	403
150	26	4767	452	3969	(5293)	8406	(11585)	594
175	34	6415	653	5766	(7728)	14172	(19313)	907
200	40	8488	982	7562	(10148)	21734	(29461)	1241
225	48	10468	1248	9594	(12936)	31328	(42397)	1666
250	56	12598	1617	11875	(15946)	43203	(58343)	2187
275	64	14654	1882	14078	(18794)	57281	(77137)	2706
300	72	16873	2261	16141	(21476)	73422	(98613)	3323
325	79	19122	2593	18578	(24610)	92000	(123223)	3934
350	86	21444	2907	21093	(27505)	113093	(150728)	4601
375	96	23972	3235	24032	(31054)	137125	(181782)	5504
400	103	26108	3657	26656	(33835)	163781	(215617)	6294
425	111	28753	4041	29187	(35581)	192968	(251198)	7213
450	119	30986	4462	32188	(37737)	225156	(288935)	8110
475	129	33571	4939	35281	(40444)	260437	(329379)	9444
500	135	35911	5435	37781	(43319)	298218	(372698)	10353

Table 1. Performance Test Results

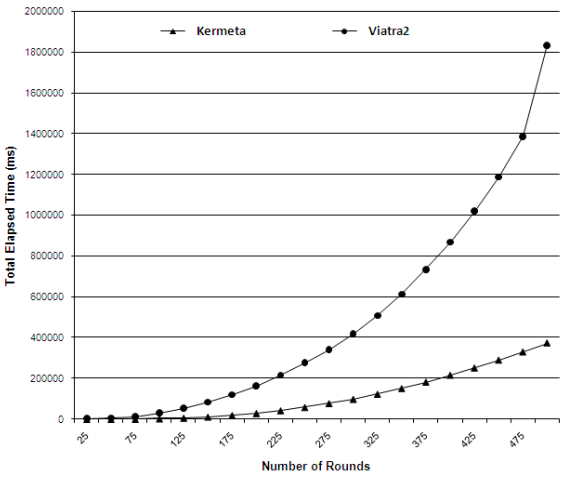

Fig. 9. Execution Time of Kermeta and VIATRA2 VDIs

6 Related Work

Several approaches can be adopted for writing model transformations. Developers can use general purpose languages such as Java, C#, and C++ or dedicated rule-based languages such as ATL [10] and the OMG QVT standard [25]. These dedicated languages mix imperative and declarative constructs. Another category of approaches include pattern-based transformation tools such as AGG [29], VMTS [18], VIATRA2 [31], and PROGRES [32].

These approaches present advantages and drawbacks. Developers who use general purpose languages benefit from well-known languages such as Java/EMF and dedicated and mature development environments such as the JDT under ECLIPSE. However, these languages are not always suitable to specify transformations because developers need to manage different constructs such as tree traversals and object instantiations. This implies a bad separation of concerns and therefore, decreases the reuse and maintainability of transformations.

In rule-based languages, simple transformations are easy to write because they are based on a one-to-one mapping that specify how to map concepts from one metamodel to another. Moreover, the declarative nature of such languages allows developers to benefit from a good expressive power without managing the rule dependencies themselves. This expressive power occurs also in pattern-based transformation tools. These languages generally support built-in static analysis capabilities such as critical pair analysis and sequential dependency analysis, which help to detect inconsistencies and implicit dependencies in transformations [15]. However, the specification of complex transformations using a declarative style can be complicated because there is no clear mapping among the concepts.

Kermeta is a domain specific language for metamodelling. It leverages object-oriented languages (Java, C#, C++, ...), class diagrams, and design-by-contract to make metamodelling easy for the seasoned object-oriented programmers. Compared to Java, Kermeta provides model-oriented and aspect-oriented capabilities: OCL-like lexical closures, native support of open-classes, model typing feature, and the ability to load and save EMF models. EMFSCRIPT [30] and EPSILON [13] share some common features with Kermeta such as OCL-like lexical closures and imperative style of programming for manipulating models. Kermeta as a model-transformation engine raises also several drawbacks. Compared to a general purpose language, the developer has to learn a new language and the environment is not as mature as an ECLIPSE JDT. Compared to a rule-based language, its imperative nature forces the developer to manage lots of concerns (tree traversal algorithms, object instantiations, ...). However, the imperative style offers more control for manipulating transformations. Finally, even if Kermeta provides a support for model transformation testing and a type checker, it does not provide any static analyzer that helps designers to detect inconsistencies in transformations. However, Kermeta still appears as a very good trade-off between general purpose and rule-based languages as well as efficient for large scale MDE applications.

7 Conclusion

In this paper, we have presented Kermeta, an executable metamodelling language for describing the structure of metamodels and their behavior. Kermeta serves as a general purpose metamodelling language that helps in solving model- and graph-based problems. We highlighted the performance, genericity, and extensibility of the Kermeta language through three cases studies. We showed that Kermeta allows the specification of generic refactorings, the partial model completion, and the efficient execution of resource consuming algorithms. Future work includes increasing the repository of refactorings on other metamodels, automatically transforming a subset of OCL to ALLOY, and optimizing the source code generated for improving the performance.

Acknowledgements. We are grateful to Vincent Mahé and Didier Vojtisek for their valuable comments on this paper and their contribution on the implementation of the Kermeta workbench as well as on the solutions of the case studies.

References

1. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In *MoDELS*, pages 436–450, 2007.
2. Enrico Biermann and Claudia Ermel. Antworld simulation case study modeled by emf transformation. In *Proceedings of the* 4^{th} *International Workshop on Graph-Based Tools (GraBaTs 2008)*, September 2008.
3. Kim B. Bruce and Joseph Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 20:50–75, 1999.
4. Sebastian Buchwald and Moritz Kroll. A grgen.net solution of the antworld case for the grabats 2008 contest. In *GraBaTs*, September 2008.
5. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the* 15^{th} *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–145, 2000.
6. Martin Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1^{st} edition, June 1999.
7. Fujaba. University of paderborn. http://wwwcs.uni-paderborn.de/cs/fujaba/.
8. Berthold Hoffman, Javier Pérez, and Tom Mens. A case study for program refactoring. In *GraBaTs*, September 2008.
9. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, April 2006.
10. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.
11. Kermeta. http://www.kermeta.org/.
12. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the* 11^{th} *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, June 1997.
13. Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The epsilon transformation language. In *Proceedings of the* 1^{st} *International Conference on Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063, pages 46–60. Springer, 2008.
14. Y. S. Mahajan and S. Malik Z. Fu. Zchaff2004: An efficient sat solver. In *Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542.*, pages 360–375, 2004.
15. Tom Mens, Gabi Taentzer, and Olga Runge. Analysing Refactoring Dependencies using Graph Transformation. *Software and Systems Modeling (SoSyM)*, pages 269–285, September 2007.
16. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.
17. Tamas Meszaros, Istvan Madari, and Gergely Mezei. Antworld, vmts. In *GraBaTs*, September 2008.
18. Visual Modeling and Transformation System (VMTS). http://vmts.aut.bme.hu/.
19. Olaf Muliawan, Bart Du Bois, and Dirk Janssens. Refactoring using jdt2mdr. In *GraBaTs*, September 2008.
20. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MODELS/UML*, volume 3713, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.
21. Niklas Een and Niklas Srensson. Minisat a sat solver with conflict-clause minimization. In *SAT*, 2005.
22. OMG. Mof 2.0 core specification. Technical Report formal/06-01-01, OMG, April 2006. OMG Available Specification.
23. OMG. OMG Home page. http://www.omg.org, 2007.
24. OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07, 2007.
25. OMG. Mof 2.0 query/view/transformation specification. Specification Version 1.0, Object Management Group, April 2008.
26. Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining mullti-formalism knowledge to select test models for model transformaion testing. In *IEEE International Conference on Software Testing*, Lillehammer, Norway, April 2008.
27. Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Domain-specific model editors with model completion. In *In Proceedings of MPM Workshop associated to MoDELS'07*, Nashville, TN, USA, October 2007.
28. Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.
29. The Attributed Graph Grammar (AGG) System. http://user.cs.tu-berlin.de/ gragra/agg/.
30. Christophe Tombelle and Gilles Vanwormhoudt. Dynamic and generic manipulation of models: From introspection to scripting. In *MoDELS*, volume 4199, pages 395–409, 2006.
31. Viatra2. Department of measurement and information systems, budapest university of technology and economics. http://www.eclipse.org/gmt/VIATRA2/.
32. Erhard Weinell. Using progres for graph-based program refactoring. In *GraBaTs*, September 2008.
33. Albert Zündorf. Antworld. In *GraBaTs*, September 2008.