# An Approach to use Executable Models for Testing

Michael Soden and Hajo Eichler

Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
[soden,eichler]@ikv.de

**Abstract.** This paper outlines an approach to test programs by transforming them into executable models. Based on OMG's metamodelling framework MOF in combination with an action language extension for the definition of operational semantics, we use QVT to transform abstract syntax trees as code representations into executable models. We argue that these models provide an adequate abstraction for simulation and testing, since platform dependencies can be resolved in a controlled way during transformation to detach the program logic from its environment. A prototypic implementation based on eclipse EMF underpins the approach.

## 1 Introduction

Execution and simulation of models are well established techniques in software engineering for decades now. While the idea of model-driven architectures (MDA) as proposed by the OMG has been successfully applied to various domains and especially embedded systems, the major part of today's enterprise software systems are *still not* developed in a model-driven way by means of transformations, integrated tool landscapes, rich traceability and 100% code generation. We identified two main reasons for this:

1. Most development languages provide similar abstraction mechanisms than models and come with considerable tool support at the code level
2. Strong execution platform dependencies of the developed code such as library or framework functionality

Worse than this is that the *meaning of models* is typically defined through the mapping into the target environment by code generators. Hence, code generators must be considered to be part of the specification when it comes to (automated) testing between the specification model and the code.

To address these problems, we have created a MOF [1] based framework that supports the definition of *operational semantics* in metamodels to precisely specify the execution semantics of models [2]. Based on the assumption that these metamodels reflect the correct platform behaviour, simulations and tests of the developed code can be executed in the model environment instead of testing it

directly on the target platform. Execution model building is achieved through a transformation defined as QVT relations [3] of a syntax oriented tree metamodel which is close to a language's EBNF grammar (similar to [4]) to an appropriate metamodel for the behaviour definition. Thereby, this import mechanism ensures to reach the proposed abstraction between model and code, which is one of the key ideas of MDA.

## 2 Execution of behavioural models

In order to execute models a (meta-)modelling framework is required that supports the definition and execution of models. For this purpose we use OMG's metamodelling framework MOF [1] in conjunction with OCL [5] and QVT [3] to precisely define and manipulate models in an object oriented way.

Even though MOF defines an overall framework for the definition and management of (meta-)models, it lacks support for the definition of concrete syntax and computational semantics [1]. To fill this gap, we extend the MOF with an action language to support the specification of *operational semantics*[2]. Through the addition of a subset of UML Actions in combination with OCL expressions, the metamodel definitions become machine interpretable and hence models executable. To clearly separate the non-changing model from its runtime configurations which evolve over time, an explicit *instanceOf* relation is introduced at M3. This explicit instanceOf modelling reduces any overhead in managing relations between (logical) classifiers and their instances. For this purpose, the instanceOf concept is aligned with a create operation which takes care of handling the creation of corresponding links to specified meta-objects.

### 2.1 MOF Actions

We briefly outline the action language in the following along with the sample metamodel of C# used throughout this paper. For a small and complete example refer to [6]. Figure 1 shows a small excerpt of the C# metamodel. The main structural part is conceptually aligned with the UML2 infrastructure library [7], although some minor modifications and simplifications have been applied (e.g. generalization is restricted to single inheritance, some associations are bidirectional, etc.) Rather noteworthy is the addition of language specific concepts such as expressions or delegates. Those parts which are only syntax variations or "syntactic suger" like property accessors, different kind of loops, etc. are represented by unified concepts in the metamodel.

The operational semantics of the runtime model is described with an action language that is syntactically borrowed from UML Actions/Activities [8]. Figure 2 shows the operational specification of the `CSAssign` meta-class as a sequence of three actions: (1) a *OCL query* retrieving the right hand side of the assignment in the context of the object `self` (which is an instance of `CSAssign`), (2) an *invocation action* that is capable of evaluating the expression and (3) a primitive, single-valued *set* action for the result. Each action is guaranteed to be atomic,

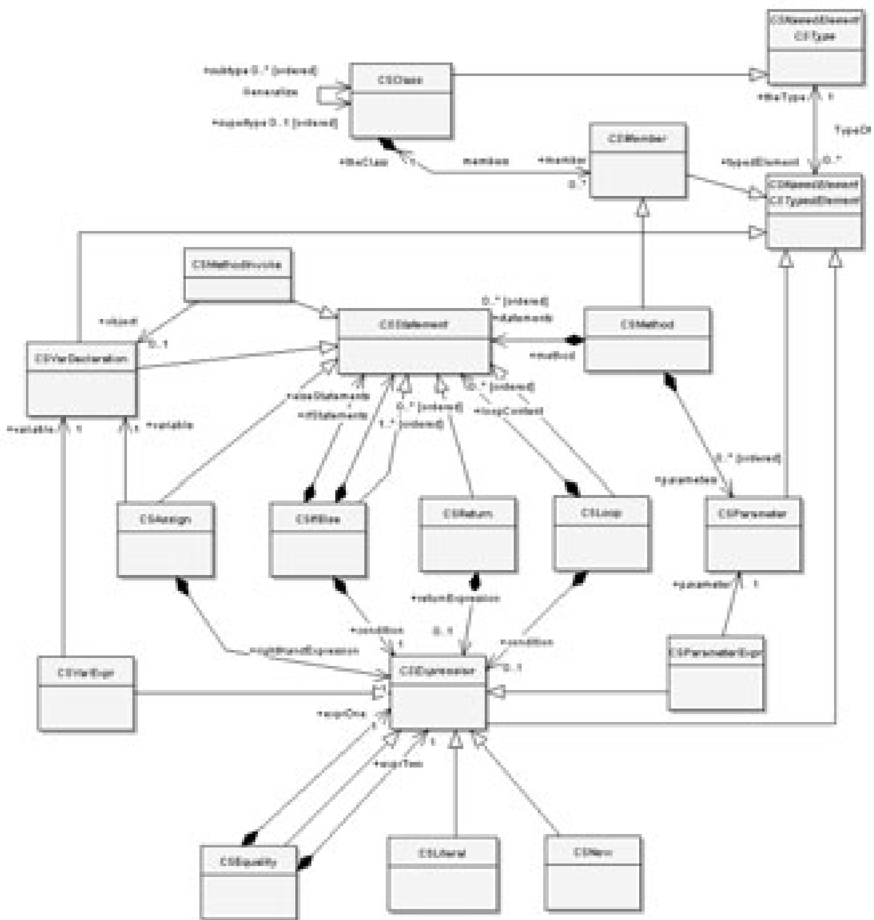

**Fig. 1.** Excerpt of the C# metamodel: structural parts and expressions

especially queries collecting elements will not be interrupted or interfere with parallel changes applied to the model[1]. Note that `self` in the query and assign actions refers to the owning `CSAssign` object while `self` at the input pin of the invocation action defines the (nested) context for the execution of the `Evaluate` behaviour.

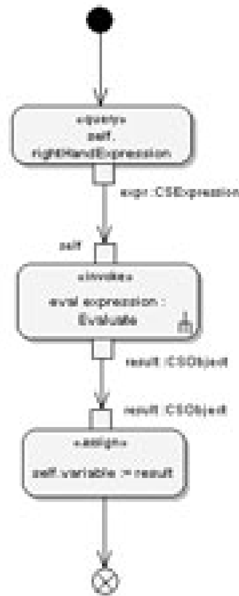


**Fig. 2.** Behaviour of class CSAssign

Beside the abstract syntax part of the metamodel, the C# runtime model is defined by specific runtime classes (cp. Figure 3). We argue that the runtime model can be regarded as an *instance of* a language's structural part. For this purpose, the *instanceOf* relationship is introduced at the M3 level to adequately provide support for "logical" multi-metalayer modelling. As consequence, each meta-object has an additional `metaObject` property that points to the specified meta-class. Existing OCL reflection capabilities such as `allInstances` or `oclIsOfType` remain valid and are still bound to the "physical" meta-layering.

Runtime objects can be instantiated with a *create* action. For example, figure 4 ("Create Method Parameter") shows a behaviour defined in the context of the `CSMethodInvoke` class. It handles allocation and binding of values for all parameters. Note that the type of the input pin of the create action is `CSParameter`

[1] Hence, there is a global order of all actions executed. Nevertheless, mutual references and modifications to shared objects are allowed

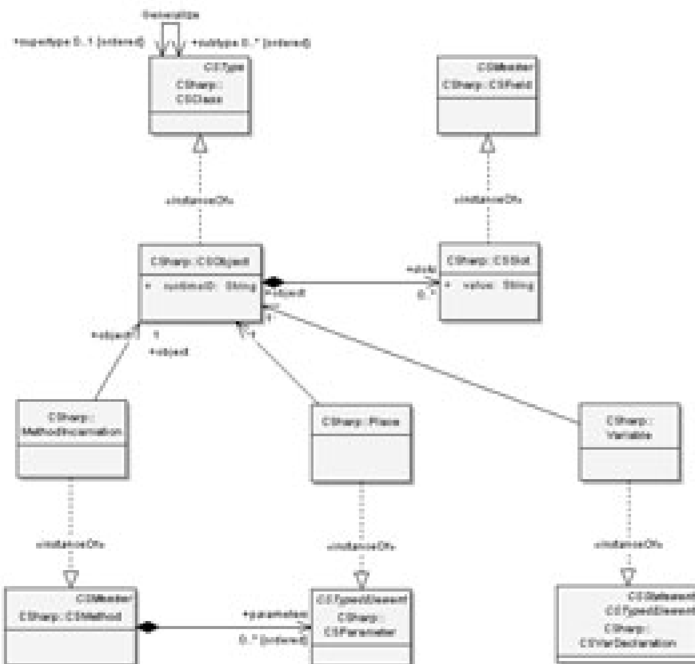

**Fig. 3.** C# Runtime Model

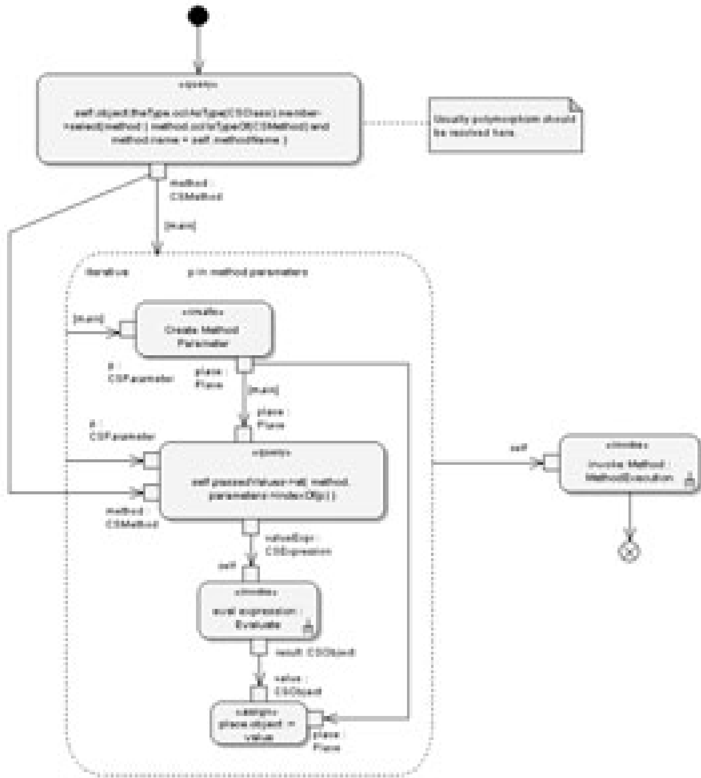


**Fig. 4.** Actions to specify method invocation

while the output pin is `Place`. Invoking this behaviour causes an object of type `Place` being created as logical instance of class `CSParameter` with a `metaObject` reference set to the `CSParameter` object passed to the input pin.

## 3 Execution of code as model

The techniques described above for designing metamodel behaviour are the basis for executing models. Figure 5 outlines the approach to analyse existing implementations in its model representation. The left side of figure 5 outlines the standard MDA approach of model to model transformation. At a certain stage the model is enriched with enough behavioural information to support model execution. The code is transformed into the abstract syntax tree using the generated

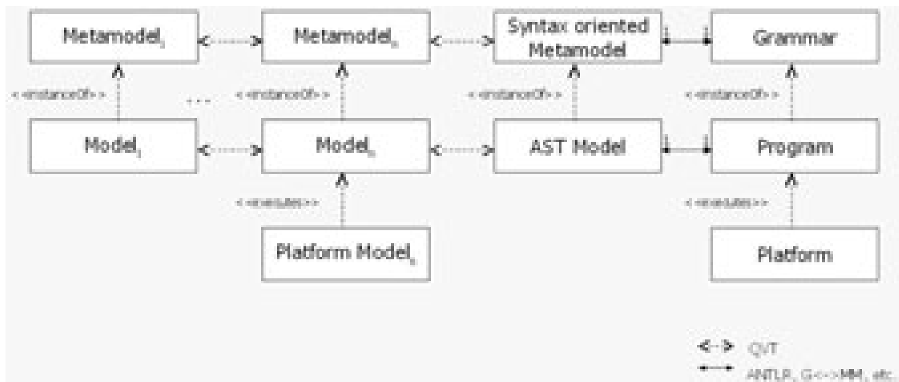


**Fig. 5.** Mappings between code, grammar and models

parser of ANTLR [9]. This grammar-based representative of the implementation will be mapped to an instance of the syntax oriented metamodel; a one to one mapping to connect the grammar with the modelling elements (compare [4]). The main difference of both representation is the data structure used. Whereas ASTs are defined by a set of independent token types with simple parent/child relation, metamodels offer in addition the advantages of object orientation like inheritance and other modelling techniques like containment. This conversion from grammar to model enables one to apply model transformations on the code representative, but it does not have any effect on the detail degree of the implementation information.

A second mapping transforms the implementation's model into the actual domain specific metamodel. One example of such a metamodel can be found in chapter 2. With this step the goal of abstraction will be achieved by two kind of mechanism. The program itself is abstracted by the mapping to its simplified model. For instance, in the model only one *iterate* definition is defined, whereas

the syntax of the language supports *for*, *while*, *do* etc. loops. The second aspect of abstraction happens by focusing on the program logic itself and extract it from it's surrounding environment. The mapping between the language and the domain specific metamodel is built using QVT relations. The following two examples show a structural and behavioural mapping between the two metamodels. One major reason to use QVT relations here is the possibility for bidirectional transformation that could ensure the re-generation of code from the model in case the model is changed. `Class2Class` as shown in figure 6 defines the mapping of the AST representation of classes to their counterpart in the domain specific metamodel. The patterns of the rule are very elementary to match all occurrences, whereas the classes content is covered by separate rules, which rely to this relation via their *when* clauses as precondition. Large part of the model

```
-- map each class to a class
top relation Class2Class
{
    varName: String;
    enforce domain ast gClass:Class {
        children = qualIdent : QualIdent {
            children = ident : Ident {
                name = varName
            }
        },
        parent = p : TreeNode {}
    };
    enforce domain metamodel mClass:CSClass {
        name =  varName,
        scope = namespace : CSNamespace {}
    };
    when {
        Namespace2Namespace(p, namespace);
    }
}
```

**Fig. 6.** QVT rule to map grammar class elements to their model correspondent

transformation forms the behavioural part. One excerpt is the rule *While2Loop*, which expresses the mapping between a *while* control flow statement and the general loop model.

## 4 Related Work

There are many frameworks for model- or language-driven development, development of DSLs, or simulation frameworks with quite different terminology. Metamodelling frameworks or tools include GME[10], XMF[11], Kermeta[12], AToM3[13], MetaEdit+[14], AMMA[15] or MPS[16]. As classification of the different approaches by means of support for the definition of structure, static constraints, representation (syntax) and behaviour (execution semantics) as done by Nytun et.al in [17], we can further distinguish two different approaches to semantics. On the one hand semantics are defined by mappings of models onto

```
relation While2Loop {
    enforce domain ast while:WhileStat {
        children = exp : Expression {},
        children = gStatements : Statements {},
        parent = p : TreeNode {}
    };
    enforce domain metamodel loop:CSLoop {
        condition = c : CSExpression {},
        loopContent = mStatements : CSStatement{},
        eContainer = container : CSElement{}
    };
    when {
        ContainmentMapping(p, container);
    }
    where {
        Statement2Statement(gStatements, mStatements);
        Expression2Expression(exp, c);
    }
}
```

**Fig. 7.** QVT rule for mapping a while statement to the loop model element

different languages or mathematical formalisms (semantic domains) as in GME and AToM3. On the other hand XMF, Kermeta, AMMA and MPS use specific action languages to define operational semantics. The approaches taken in XMF with XOCL (eXtensible Object Command Language), Kermeta's textual action language with OCL and QVT all have in common that querying is achieved by OCL's navigation capabilities. This idea is reused in our approach. Additionally, the work of [18] inspired us to express the operational semantics with a reduced set of UML actions. However, controlling atomicity of composed actions is rather comparable to the "step" keyword of the Abstract State Machine Language (AsmL[19]). In the same way as such ASMs define the formal semantics of e.g. the SDL specification[20], our actions follow a similar approach but replace evolving algebras with manipulations of runtime configurations that are instances of MOF metamodels.

Although instantiation is at the core of any metamodelling facility, the approaches differ in their realisation in the frameworks. We argue that while the abstract syntax model is logically at M1, the runtime configurations are located at M0. Atkinson et al.[21] analysed the shallow/deep-instantiation and strict/non-strict metamodelling approaches and pointed out the *ambiguous classification problem* and the *replication of concepts* problem. However, we argue that explicit (shallow) *instanceOf* modelling helps to distinguish multiple logical meta-layers within the concept space defined by a metamodel.

## 5 Discussion

Our approach addresses the problem of decoupled working on model and code level, whereby models do also have a behavioural description of the underlying platform. To execute code as model for testing purposes a couple of advantages against traditional techniques can be found. First, within the abstraction also a major aspect for simplifying testing is found. On the one hand, concentrating on the logic of the program is also a key for writing tests/execute models on the problem scope. On the other hand, it is possible to derive the counterparts of

so called *mock objects*, which emulate a part of the system which is irrelevant for the actual component under test, on importing the code to the model. For example, typical three tier architectures based on the Model View Controller paradigm often requires a lot of code for test-drivers and mock- objects on the GUI and database level. Even though we succeded only in small replacements of console based input/output, bigger replacements of library functionality could be applied easily. Another important aspect is that execution in the model will lead to scenarios containing model data: the actual testing data. Those scenarios could be recorded and reused for testing.

## 6 Conclusion

The paper outlines an approach to translate existing implementations into their corresponding domain models in order to execute and test their behaviour. The behaviour is defined through an action language extension of MOF that supports the definition of operational semantics. Utilising this approach helps testing the actual implementation by abstracting from the language's concrete environment. Thus, it supports testing and simulation of the implementation decoupled from the platform and other specific library or framework functionality. Our current results are promising that often faced overhead of building test-stubs, simulating network capabilities in test-drivers or omitting GUI references can be solved all at once.

A prototypic implementation has been carried out based on eclipse EMF [22] and a QVT engine [23]. Further directions for the implementation are towards recording of test-runs and comparison against previously executed simulation runs to really test the developed application against its specification.

## References


1. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group (2003) ptc/03-10-04.
2. Plotkin, G.: A structural approach to operational semantics. Technical report, University of Aarhus, Denmark (1981)
3. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Draft adopted Specification, ptc/05-10-02. Object Management Group (2005)
4. Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Tucs technical report no 606, Turku Centre for Computer Science (2003)
5. OMG: OCL 2.0 Specification. Object Management Group (2005) ptc/2005-06-06.
6. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. (2007)
7. OMG: UML 2.0 Infrastructure Specification. Object Management Group (2003) ptc/03-09-15.
8. OMG: UML 2.0 Superstructure Specification. Object Management Group (2004) ptc/04-10-02.

9. Parr, T.: (ANTLR – Another tool for language recognition) Last checked: February 8, 2006.
10. Agrawal, A., Karsai, G., Ledeczi, A.: An end-to-end domain-driven software development framework. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 8–15
11. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodeling, A Foundation for Language Driven Development. Xactium (2004)
12. Team, T.: (Triskell Meta-Modelling Kernel. IRISA, INRIA. www.kermeta.org.)
13. : (The Modelling, Simulation and Design lab (MSDL), School of Computer Science of McGill University Montreal, Quebec, Canada: AToM3 A Tool for Multi-Formalism Meta-Modelling. http://atom3.cs.mcgill.ca/index.html.)
14. MetaCase: (MetaEdit+. http://www.metacase.com.)
15. Davide Di Ruscio, Frric Jouault, I.K.J.B.A.P.: Extending amma for supporting dynamic semantics specifications of dsls. Technical report, Universitegli Studi dell'Aquila (2006)
16. Dmitriev, S.: Language oriented programming: The next programming paradigm. onBoard (1) (2004)
17. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based language development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (2004)
18. Sunyé, G., Guennec, A.L., Jézéquel, J.M.: Using uml action semantics for model execution and transformation. Inf. Syst. **27**(6) (2002) 445–457
19. Yuri Gurevich, Benjamin Rossman, W.S.: Semantic essence of asml. Technical report, Microsoft Research (2004)
20. ITU-T: SDL formal definition: Dynamic semantics. In: Specification and Description Language (SDL). International Telecommunication Union (2000) Z.100 Annex F3.
21. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. LNCS, London, UK, Springer-Verlag (2001) 19–33
22. Eclipse Project: Eclipse Modeling Framework. (2006) Last checked: January 1, 1970.
23. : medini QVT Engine. (www.ikv.de)