

# A Meta-Modelling Framework for Modelling Semantics in the Context of Existing Domain Platforms

Hajo Eichler<sup>2</sup>, Markus Scheidgen<sup>1</sup>, and Michael Soden<sup>2</sup>

<sup>1</sup> Department of Computer Science, Humboldt Universitt zu Berlin  
Unter den Linden 6, 10099 Berlin, Germany  
scheidgen@informatik.hu-berlin.de

<sup>2</sup> IKV++ Technologies AG  
Bernburgerstrasse 24-25, 10963 Berlin, Germany  
{eichler,soden}@ikv.de

**Abstract.** Models based on meta-models have been recognized as essential assets in model driven software engineering. OMG's meta-modelling framework MOF allows the definition of languages as object-oriented structure models, but it does not allow to specify the semantics of languages. This has three reasons: The inability to meta-model the behaviour of the modelled concepts, missing possibilities to influence the instantiation of runtime elements at meta-level (shallow instantiation), and environment interactions can only be realized by means of model transformations.

We propose a modelling framework that uses action semantics to model concept behaviour. We distinct between logical and physical instantiation concepts to describe interactions between models and runtime elements at meta-level. Furthermore and in contrast to other semantic modelling environments, we incorporate "real-world" M0-layer elements to allow interactions between models and existing domain platforms, thereby allowing early model-based simulation and testing, embedded in concrete environments.

## 1 Introduction

Model driven development uses models as main asset to model a system at different levels of abstraction; it uses mappings to define relations between different elements of different models. The modelling languages used in such a process have to be defined in a formal way that allows the definition of mappings between them. OMG's Model Driven Architecture (MDA) characterises two different abstraction levels; the models on those levels are platform specific (PSM) and platform independent models (PIM). To analyse a system using abstract platform independent models, MDA requires modelling languages that allow execution (simulation) of models in the absent of a concrete environment.

An important application of model driven development techniques is domain specific modelling. It requires languages that allow a domain expert to model

a system with domain specific concepts. Such domain specific languages must often work in certain existing environments, because most domains already use computer based systems, systems that a new domain specific language has to extend. Domain specific modelling requires efficient language development, because it requires highly specialised solutions for small markets, and a language development that allows swift integration of existing domain specific environments.

Both model driven development in general and domain specific modelling in particular need language definitions that not only define the syntax of language, but allow modelling of language semantics and interactions with environment as part of those semantics. Today's development of such language definitions is mostly meta-model based. Concrete syntax and semantics rely on a shared abstract syntax meta-model that represents the static part of the language definition. Semantics are usually realized either by one of the following three methods: (1) using a programming language and a meta-modelling framework, for example EMF [1]. A more sophisticated approach (2) is to use a framework that allows semantics representation by mapping on a well defined target language, like GME [2] does. Or (3) you use a method that allows to define your own semantic domain. But this method usually does not allow interaction with existing environments, e.g. MMF approach [3] or later XMF [4].

What one actually needs in an approach that: (1) does not rely on programming on a concrete platform and all its disadvantages, (2) does not simply translates semantics of one language to the semantics of a target language and therefore relies on existing languages, and (3) does not only support closed definitions that are hard to use in conjunction with existing environments. In this paper we present an approach based on established techniques: Extended meta-modelling that allows structural modelling of language concepts (syntax) and runtime concepts (semantics), UML activities based on Actions and OCL to describe the behaviour of concepts, and automated domain model generation that allows to define environment interaction at meta-level.

The next section will present related work, and shows what distinguishes this approach from existing methods. We will then introduce a simple domain specific example language in section 3. Then three sections will follow that explain the details of the approach: Structural modelling of syntax and semantics in one model in section 4, modelling the behaviour of language concepts in section 5, and how existing domain environments can be integrated into language definitions in section 6. The paper ends with the concluding section 7.

## 2 Related Work

### 2.1 Meta-modelling different modelling layers

Common meta-modelling architectures are based on the 4-layer (M0-M3, bottom up) paradigm; each layer defines the elements used on the layer below. An even stronger meta-modelling principle is that each element is defined by exactly

one element from the next layer above (*strict meta-modelling*). In [5] Atkinson and Kühne introduce the main problem with this way of meta-modelling known as *shallow instantiation*: Since an element does only classify its own instances (an M2-element models M1-elements) it cannot influence the instances of the element it describes (an M2-element cannot describe M0-elements). This means that at the level of language definition (M2) one can describe the structure of language concepts (abstract syntax, M1) but one cannot describe the structure of the instances of those concepts (semantics, M0).

They propose different, as they say unsatisfactory, solutions to the problem. One solution, *Nested Metalevels* uses two different meta-models (M2-models) to describe both (M1-elements) and (M0-elements) at the meta-model layer. Alvarez et al apply this solution to the MMF approach [3] in [6]. Here the authors describe the relations between model and object layer (M1 and M0) as a mapping between syntax models (models of M1 elements) and models for semantic domains (model of M0 elements).

Atkinson and Kühne [7] distinct between a logical and a physical meta-modelling dimension, to allow different strict instantiation hierarchies to coexist. We will use this distinction to map logically strictly modelled M1 and M0 layers to one physical layer. In result the physical dimension shows the usage of the *Nested Metalevel* solution, but the layers form at least logically a strict 4-layer architecture. In contrast to Alvarez and Clark et al, we use instantiation semantics based on a one-to-one *InstanceOf* relation to map M1 elements to M0 elements instead of using a custom mapping function. Instantiation becomes a part of the behaviour definition for language concepts, and a distinct mapping between abstract syntax and semantics is unnecessary.

## 2.2 Semantics definition for meta-model elements

There are two possibilities to define the semantics of a meta-model element. One is to define a behaviour that describes what a instance of the meta-model element means, e.g. how it behaves at runtime. The other is to map meta-model elements to other M2-layer elements that already have defined semantics.

Engels et al [8] use UML collaboration diagrams to define the behaviour of meta-model elements by means of operational semantics. A model written in the described language is used to instantiate the abstract syntax elements. After instantiation their behaviour is invoked. Thus, defining the behaviour of the model itself. Muller et al emphasize the importance of OCL in such approaches. In their own simple programming language, used to define the behaviour for operations of meta-model elements, OCL is used to navigate the model. The language itself uses only a very few language constructs, while it still achieves high expressiveness by utilizing OCL's ability to easily query the model.

The transformation based approach is used by Agrawal et al [2] and Chen et al [9]. Using their method called GME, a language cannot be defined in itself; a suitable semantic domain must already exist. The MMF approach [3] or later [4] can be classified as a hybrid. It uses mappings between abstract syntax models

and semantic domains, and allows to specify semantic domains with either operational semantics (using an extended OCL that allows side effects, e.g. model modifications) or mapping to other semantic domains.

OCL seems to be a necessity for semantic definition, since it is easy to write and understand, it is well known, and allows complex queries over models at a high level of abstraction. No matter whether it is used to define the semantics of meta-model elements as instance behaviour or is used to define mappings between abstract syntax and semantic domains, OCL should be used in a semantic meta-modelling framework.

Action semantics, in contrast to collaborations [8] or imperative programming languages [10], allow behaviour definition with simple activities based on an extensible set of actions. Since actions can be defined in the physical dimension and thus be used for both logical M1 (abstract syntax) and M0 (semantic) models, the same language can be used to define the behaviour for both logical layers. The activity language itself is well known from UML activity diagram, and easy to apply for both the behaviour of classes (meta-model classifiers) and operations (behavioural features of meta-model classifiers). Other advantages of defining meta-model semantics with behaviour models rather than the use of transformations will become clear from the next sub-section.

### 2.3 Language integration in existing platforms

The standard way to let an abstract specification interact with a platform is to map the abstract specification onto a concrete platform. This is the core principle of the MDA method. Chen et al apply this principle to the definition of languages in [9]. They use their meta-modelling framework GME to map domain language concepts onto existing and semantically well defined concepts in the target language, e.g. Abstract State Machines (ASM) [9].

A different approach is to use domain models. That is in contrast to map language concepts to concrete platform concepts (lower the level abstraction), to create an abstract model for the domain (rise the level of abstraction). This domain model can then be used to relate language concepts and domain models on the same level of abstraction, e.g. using the same meta-modelling language.

Existing work ([11]) creates domain models from abstract domain concepts using for example an ontology based approach. But the semantics of the domain concepts has still to be defined. This can be done by utilizing the fact, that for many domains, domain knowledge has already been covered in functionality offered in libraries (domain libraries) that can be accessed using APIs. Those libraries form environments that domain specific languages have to interact with. We will therefore relate the semantics of domain concepts to domain models based on existing domain libraries and thus existing domain environments.

## 3 A simple example – Traffic lights modelling language

For better understanding of the proposed approach, we introduce the traffic light example, which we use to explain the concepts of this paper in an understand-

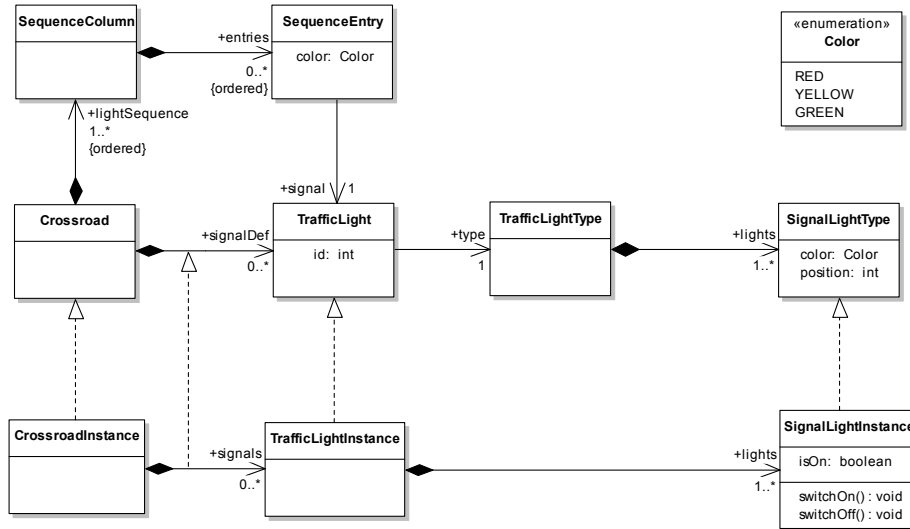


Fig. 1. Abstract syntax for a simple traffic light modelling language

able way. Figure 1 shows the meta-model of a domain-specific language to define traffic lights. On a *Crossroad* a set of *TrafficLight* is installed, whereas a *TrafficLight* is of type *TrafficLightType*. A *TrafficLightType* is a construct, which contains a number of *SignalLightType*, which are positioned lights with either a red, yellow or green colour. An example instantiation can be found in figure 2 on the left side, where two kind of *TrafficLightTypes* are defined - one for vehicles (3 lights), the other for pedestrians (2 lights). Both types are used twice on our crossroad (cp. figure 2, right). With the meta-model elements *SequenceColumn* and *SequenceEntry* the light switching of all traffic lights on a crossroad is recorded. To simplify matters, the example is reduced of timed behaviour in the switching sequence.

Imagine that this language is used to define traffic lights e.g. for normal crossroads or T-crossways, which exist many times in real world. In a potential scenario a simulation should handle a so called "green wave" of a couple of crossings along a certain way to lower traffic jam. Therefore, it will be necessary to describe the explicit instances, either in a M0 simulated world or for a test in the M0 real world.

#### 4 An additional instantiation concept

In the traffic light example we introduced three instance definitions: *CrossroadInstance*, *TrafficLightInstance* and *SignalLightInstance*. These model elements are used as the M0 representation, the real world traffic lights. The whole usage of these elements regarding to the action semantic will be shown in section 5.

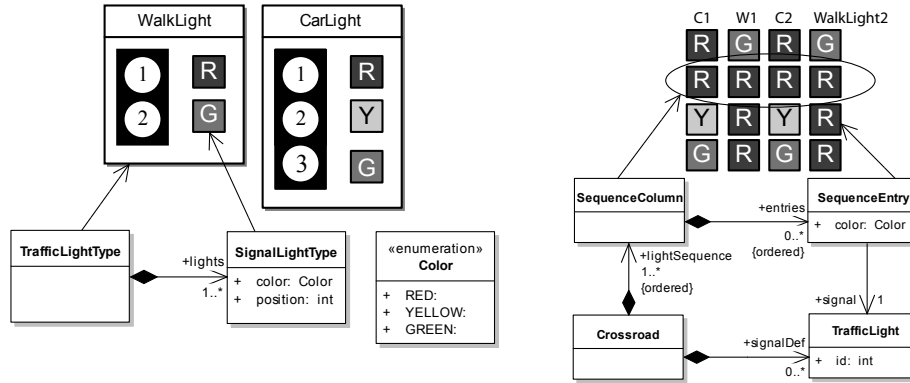


Fig. 2. A possible concrete syntax for the traffic light example

Here the embedding of the concept *InstanceOf* in the MOF is outlined. Since, M1 and M0 elements are displayed in one and the same meta-model (on layer M2) they must be exactly distinguishable to fit in the 4-layer architecture of MOF.

A way to express the relationships between the layers, e.g. between the M1 *Crossroad* concept and the M0 *CrossroadInstance* concept, is an additional relation in the meta-modelling language (cp. figure 3, lower part). This *InstanceOf* relation is (1) not only usable between MOF classes, but also between associations and (2) a meta-model with more than one *InstanceOf* relationship becomes hardly to understand, we use the suggested relation in the MOF M3 model, shown in figure 3, upper part. The concrete syntax to express a *InstanceOf* relationship is to draw a dashed generalization arrow.

The *InstanceOf* relation determinates on which level the instances of any M2 definition will be either M0 or M1. This distinction is expressed with the following OCL statements: Definition of M1: *self.classifierClassifier->isEmpty()* and *self.allParents()->forall(s : Classifier | s.classifierClassifier->isEmpty())*; definition of M0: *self.classifierClassifier->notEmpty()* or *self.allParents()->exists(s : Classifier | s.classifierClassifier->notEmpty())*. Thereby, *allParents()* is an OCL operation from the standard MOF model, which returns the set of classifiers which can be navigated recursively from the current element via the *general* property.

To define the semantics of the new *InstanceOf* relation, the framework will substitute any usage of the *InstanceOf* concept with a normal MOF association and a few supporting operations at the related classifiers (cp. figure 3, lower part). The operation *isInstanceOf(o:Crossroad)* is available in any M0 definition for polymorph type check, whereas in M1 definition the operation *conformsTo(c:Crossroad)* shadowed type queries. We propose the following standard implementations for these methods: *x.conformsTo(y)* implies  $x = y$  and *x.isInstanceOf(y)* implies  $x.classifier = y$ . Further, a create method is available

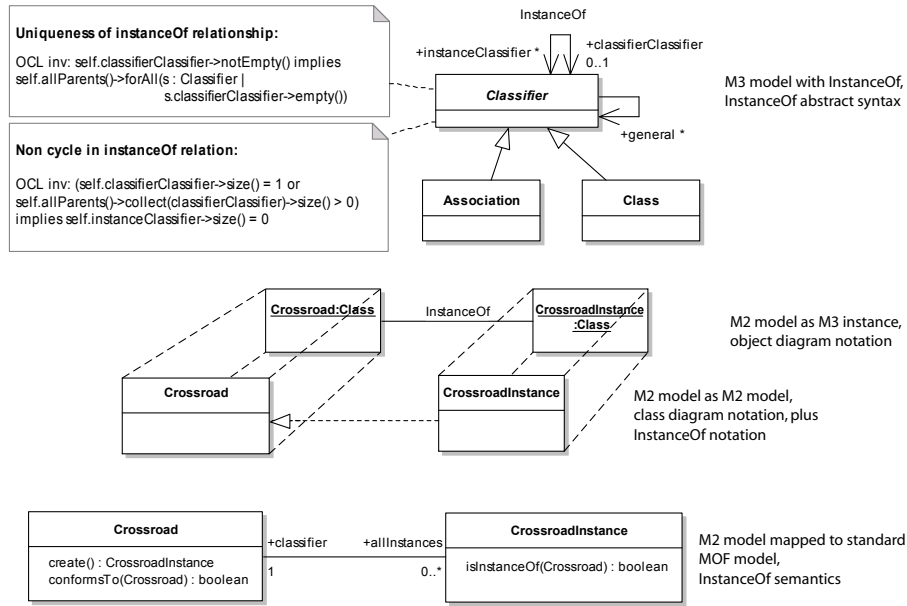


Fig. 3. Syntax and semantics for the *InstanceOf* relation

on M1 level, which will instantiate a new M0 object as well as establish the link between the newly create object and its creator, the M1 instance, the create method was invoked on.

As pointed out earlier, the *InstanceOf* relationship will also be applied on MOF associations. Thus, links between M0 level instances can be restricted regarding to the definition of a specific M1 link. This concept is well known from MOF associations defined on M2 and the link set on M1 level. We apply the same semantic on the level below to ensure consistency between M1 and M0 links.

## 5 A Behaviour Language

Our proposed meta-modelling framework is an extension of MOF as a widely accepted meta-modelling facility based on the object-oriented paradigm in combination with hierarchical meta-layering. MOF provides already concepts for the definition of the abstract syntax of a language by means of meta-models. As an extension for the definition of language semantics, we propose *MOF Action Semantics* (MAS) which reuses parts of the UML Actions and Activities with a tight coupling to OCL. The design rationale concentrates on the specification of computational semantics of languages. Taking advantage of the logical layering provided by the MOF meta-layer architecture, the combination of OCL querying with the expressiveness of UML structured activities builds the foundation

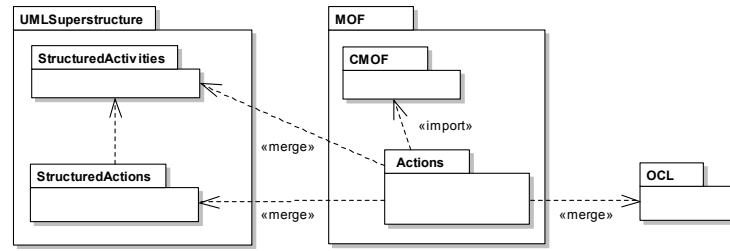


Fig. 4. The M3 model based on MOF, UML Superstructure and OCL

of powerful language for semantic definitions. Because, a full description of the semantics is not possible in the extent of this paper, we only highlight some of the sweet-spots of the language.

The definition of MAS is realized by a package-merge from the UML Superstructure and OCL meta-model into CMOF (figure 4). `MOF::Actions` merges the `StructuredActivities` and `StructuredActions` packages to inherit basic behavioural mechanisms such as control nodes, structured activity definitions, invocation actions with activity parameters, control and object flows with pins, and especially structuring capabilities, object flows and fork/join describing concurrency and process interaction make up a powerful tool as expressive as UML Activities.

Since the alignment of OCL and MOF is still an open issue our meta-meta-model has to combine the UML Actions and OCL expressions in a pragmatic way at least from an implementation point of view. To make iteration over OCL collections possible, a specific action type is introduced which provides one object of type of a collection at its output pin while iterating. Conceptually, we redefine the OCL iterate expression (class `LoopExp` of the OCL meta-model) by adding a few associations. Additionally, we introduce mainly four new action types beside the basic invocation actions inherited through the package merge: `OCLQuery`, `Assign`, `CreateScope` and `Instantiate`.

`OCLQuery` is added to allow arbitrary queries to be executed over a model and its result being used at a corresponding output pin for other actions. This is comparable to the usage of OCL in QVT for querying models. Hence, it cures typical navigation diseases by supporting arbitrary queries seamlessly over all meta-layers and `InstanceOf` relations. However, the types of OCL expressions allowed are limited to those that define the navigation capabilities. The `Assign` action has a comparable semantic to the `LinkAction` in UML, but is added because it is not obtained by the package-merge and furthermore needs alignment with the action classes. `CreateScope` is a specific action to specify a set of elements being copied for easier specification of e.g. call frames.

As already outlined in section 4 an essential enhancement addressing the shallow-instantiation problem is the instance model with its logical layering. One goal was to try eliminating intricate overhead in managing relations between (logical) classifiers and their instances. Therefore, the `InstanceOf` concept is

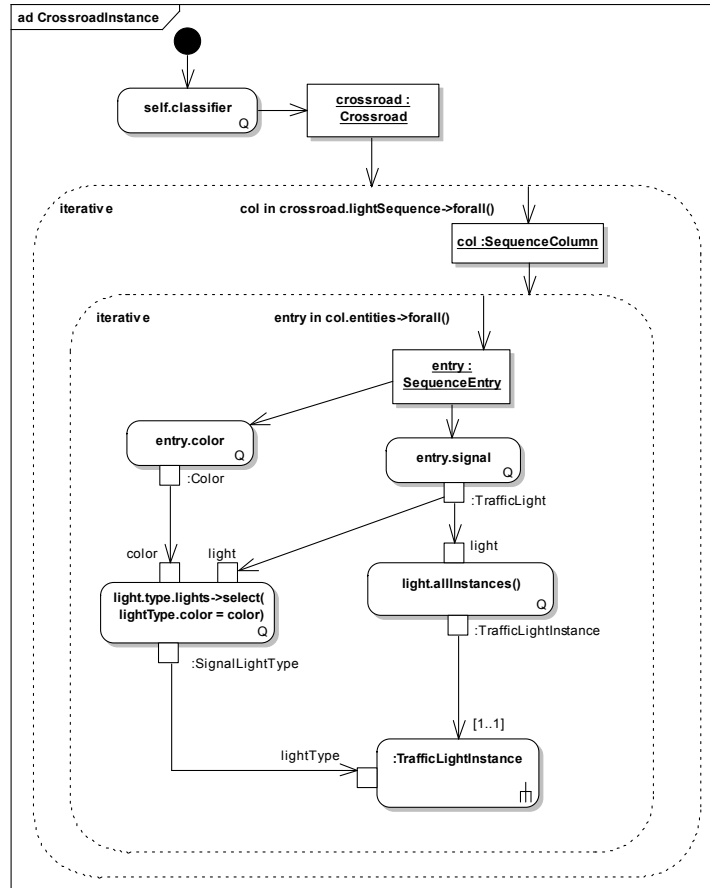


Fig. 5. A behaviour for CrossroadInstance

backed by an *Instantiate* action triggering the implicit create operation described in section 4. The default behaviour of this operation takes care of handling the creation of corresponding links to specified meta-objects taking into account associated classes that represent instantiations in a commutative manner, i.e. instances are linked if, and only if their classifiers are associated.

Having all these concepts at hand, we exemplify the definition of computational semantics on the basis of the *TrafficLight* example introduced in section 3. Figure 6 depicts the behaviour of the class *SignalLightInstance*, which states a basic on/off switching functionality. Note that *SignalLightInstance* is a classifier that has an associated activity behaviour. The first action is a simple example of an *OCLQuery* action. For the OCL part, the context of an expression is defined by the class that contains the actions directly or indirectly (if it is owned by an operation). Hence, *self* is well-defined and gives access to the contextual instance

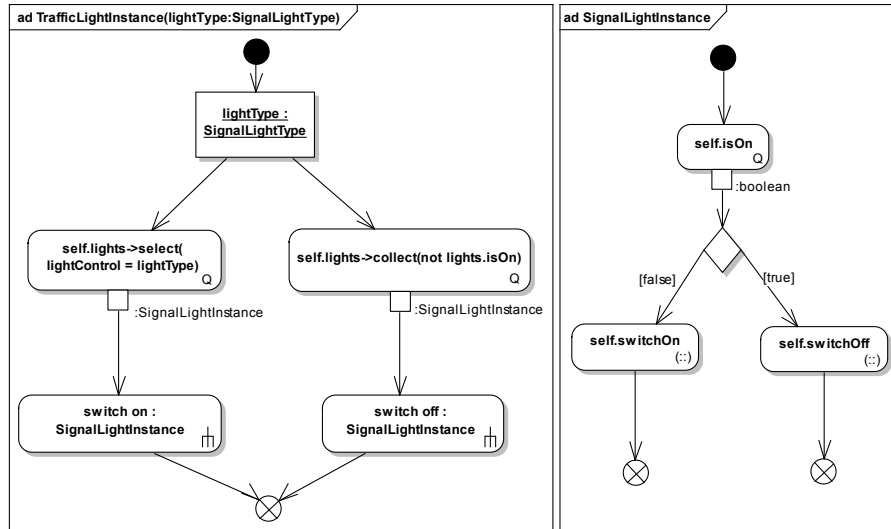


Fig. 6. Behaviours for TrafficLightInstance and SignalLightInstance

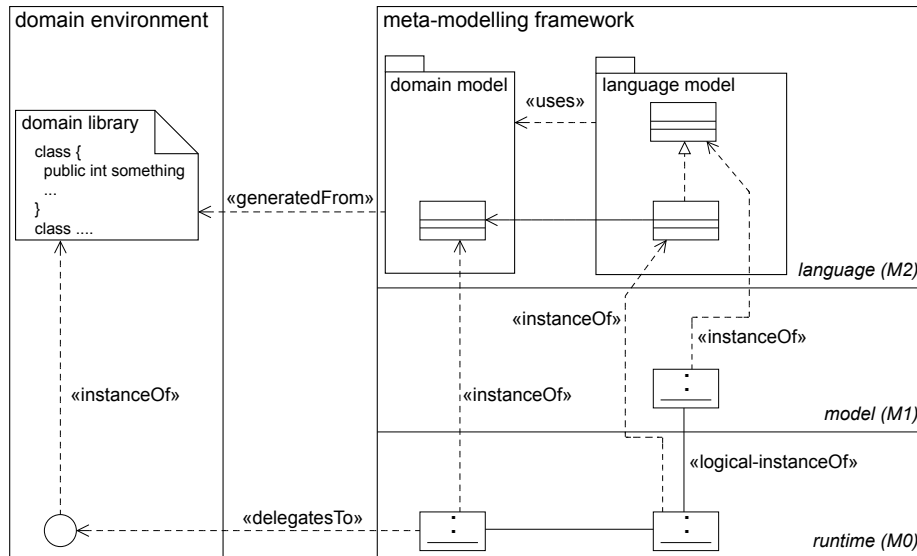
(here: of type *SignalLightInstance*). The OCL query hands over its result via the output pin to a connected decision node. Both two follow-up actions are normal operation invocations as defined by UML.

Based on these behaviour definitions, we can build the activity for the sequential switching of lights (cp. Figure 5). Here, the context of this activity flow is the *CrossroadInstance* meta-class. Basically, this control logic is built-on iterating through the light sequence defined by meta-class *SequenceEntry*. The iterate actions are notated as iterative expansion regions in terms of UML syntax, declaring their iteration expression in the form of *variable in collection*. Most of the other actions denote *OCLQuery* actions with navigation statements.

The example shows the seamless integration between OCL queries and object flows via pins, regardless of whether the return values are collections, primitives or class types. To stress the structuring capabilities, the behaviour is explicitly distributed over several meta-classes. For this reason, e.g. *TrafficLightInstance* encapsulates its execution semantic which is invoked via a behaviour call action. As in UML, parameter passing is realized by activity parameters (here: *lightType*).

## 6 Including Domain Models

Every language defines constructs to relate the models written in that language with an environment. The famous "Hello World" program, so often used to introduce to a new language, explains the interaction with the environment before



**Fig. 7.** The meta-modelling architecture in conjunction with domain models

most other language concepts. And independent of the language kind, a multi-purpose implementation language, unified modelling language, or domain specific language, the complete semantics of a language can only be fully explained in conjunction with the concrete environment that a program or model written in that language is executed in.

Such environments are usually called platforms and as an abstract concept, platform covers almost everything from simple file-IO APIs to complex middleware platforms. Especially in the context of domain specific language development, the language designer often is confronted with existing domain libraries and it is his job create a language that gives domain experts access to this functionality in a syntax that they can understand and use.

When we desire to model a language completely, we also have to relate model elements with entities that represent existing environments. Logical M0 elements that we use to describe the semantics of a language have to interact with entities outside the actual language. Whereas in MDA, this is usually done by mapping platform independent models to models that dependent on the specifics of a concrete platform, we propose a different approach that automatically creates a model-based representation of the domain environment and allows the model-based language definition to include those domain model elements in the definition of the language semantics.

Figure 7 shows the resulting meta-modelling architecture including domain environments and the according domain models. Domain specific APIs describe entities in the execution layer (M0). For those APIs a model representation is

generated. This representation can be used within a meta-modelling framework at meta-model level (M2). They can be transparently used, like every other model element, within the language’s semantics definition (also M2). When the language model is instantiated by an user model or program (model-level, M1) and the behaviour for those M1-elements is executed, it will create runtime entities (M0) according to the defined semantics. Those M0-entities will be normal model elements that live within the modelling-framework or will be instances of elements in the domain library. The latter ones are represented by model elements, but all actions taken on those elements will be delegated to the implementation in the real domain environment and will thus trigger actions in the real world.

### 6.1 A mapping from domain libraries to domain models

In most occasions domain functionality is given in a library written in an object-oriented programming language, since that was the status quo to realize domain concepts in the last 30 years. To use those functionality in model based language definitions, it is necessary to map the interfaces of domain libraries (API) to elements in the model-world. In object-oriented languages, and as a special case structured programming languages, interfaces consist of classes offering instance-scope and classifier-scope features. Those features can be properties, represented by pairs of getter and setter methods, and parameterised operations. Because meta-modelling languages, e.g. MOF, are based on object-oriented modelling, they offer all necessary concepts to easily map object-oriented APIs to meta-models.

We want to introduce a mapping for the Java programming language to MOF 2.0 meta-models. We propose a mapping that uses the same principle that MOF 1.x language mappings used, but in the reverse direction. Since MOF 2.0 models do not support classifier-scope features anymore, for each domain library type (interface, class) two MOF classes are created: One represents instances of the given type (object-proxy), it offers all instance-scope features; and one class (class-proxy) with singleton semantics that represents all classifier-scope features (static methods and properties) and offers access to functionality that creates instances of that type (constructors). Figure 8 shows a simple example. Other

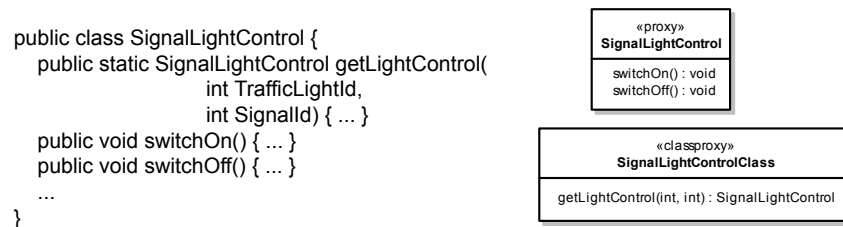
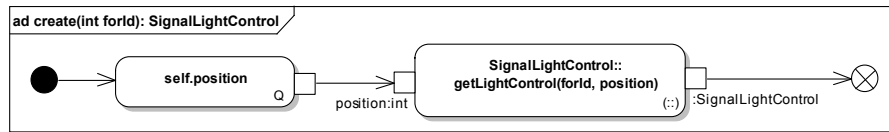


Fig. 8. Example for the mapping of an API to a domain model



**Fig. 9.** An activity to replace the standard create semantic for a custom environment object

Java types, like primitives or enumerations, are mapped accordingly. Exceptions are not supported by MOF, but can be realized by normal class-based types and throw clauses can be mapped to additional out parameters. Similar mappings can be developed for different meta-modelling languages.

After applying the mapping to a domain library, the resulting model classes can be used like any other, user generated class, except for creating instances of those classes. Normal MOF elements are created using a factory based on a model package. In the case of a domain model the factory will allow access to all class-proxy elements (thus it controls their singleton status). The factory will also allow the normal creation of object-proxy instances, if the original Java-class has a default constructor. In all other cases instantiation is only possible using the according class-proxy instance.

Since the proxy elements are basically normal model elements that are maintained through the same instance model than every other model element, domain models seamlessly integrate into the rest of the meta-modelling architecture. All actions and model queries do also work on domain model elements. Utilizing this transparency language models written without domain models, can subsequently be extended by exchange single language runtime concepts with domain model elements. The specific semantic of domain model elements becomes evident in a very low level of the meta-modelling framework. Where the instance model for regular model elements realizes all actions taken on the element within the meta-modelling framework, the instance model for domain model elements delegates all calls to the original domain library functionality.

## 6.2 The traffic light example with a real traffic light

One way to integrate the domain model into the example traffic light language is simply to exchange the original *SignalLightInstance* with *SignalLightControl*. Instead of changing the boolean attribute, the same behaviour definition will now turn on and off the real lights. The object proxy class can be used as any other class, it can be associated or be an instance of an other class (M0 or M1 element).

The *SignalLightControl* is created using the factory method pattern. There is no standard constructor to create instances using a general interface. The behaviour of the create method that corresponds to the *InstanceOf* relationship between *SignalLightType* and *SignalLightControl* has to be customized. The ac-

tivity in figure 9 defines this behaviour by invoking the factory method. Since the interface to create an instance stays unmodified, all actions, especially the create action, will still work and the original language behaviour specification does not have to be changed.

## 7 Conclusions

We implemented the proposed semantic meta-modelling framework based on the MOF 2.0 CMOF modelling framework *A MOF 2 for Java* [12,13]. Our prototypical implementations uses all explained techniques: It realizes instantiation semantics by generating associations and methods with the appropriate behaviour. The activities and actions are implemented according to the UML 2.0 Superstructure meta-model and the OSLO-OCL tools. A generator for domain models based on Java libraries exists, and the models are integrated with the according semantics into the framework.

So far we used our framework to define semantics and create simulators for small parts taken from existing programming languages (instantiation based on object-orientation) and modelling languages like SDL (state machine based execution modelling). We plan to further evaluate and asses the advantages of our method with a combined SDL and UML semantics model, working on existing runtime-environments for distributed-computing. We hope to learn more about the possibilities to reuse meta-models, including semantic descriptions, in the context of similar languages and concept specialisation. More importantly, we want to integrate the result of our research in real projects as case studies for the applicability of our method based on the constraints in industry projects.

The work presented in this paper leads to a number of questions for further research. We used the usual 4-layer model, but the method is open to arbitrary numbers of layers, but is this reasonable? What other ways of integrating a concrete environment exist, how to model based on environment features that are not described through simple programming APIs. The used action language depends on actions, what action else are reasonable? How to realize reoccurring concepts of semantic modelling in a general and reusable way, especially concepts important for execution semantics like time and concurrency? We think that many of those questions, will (have to) be answered when we continue to apply our method to real languages and real domains.

In summary, we presented a semantic meta-modelling framework that attacks three core problems of modelling the semantics of languages and gives practical solutions for them. Our solution to *shallow instantiation* can be applied to existing modelling frameworks right away; to model the execution behaviour of language concepts with UML activities and actions allows abstract modelling with known and broadly accepted means, while using OCL to keep expressiveness high; finally, do domain models allow a pragmatic way to integrate existing implementations of domain concepts easily.

## References

1. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T. J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley Professional, 1st edition (2003). ISBN 131425420
2. 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. ACM Press, New York, NY, USA (2003). ISBN 1-58113-751-6, 8–15. doi:<http://doi.acm.org/10.1145/949344.949347>
3. Clark, T., Evans, A., Kent, S., Sammut, P.: The MMF approach to engineering object-oriented design languages. In: Workshop on Language Descriptions, Tools and Applications (2001)
4. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodeling, A Foundation for Language Driven Development. Xactium (2004). URL <http://www.xactium.com/>
5. 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. Springer-Verlag, London, UK (2001). ISBN 3-540-42667-1, 19–33
6. Álvarez, J. M., Evans, A., Sammut, P.: Mapping between levels in the metamodel architecture. In: UML '01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer-Verlag, London, UK (2001). ISBN 3-540-42667-1, 34–46
7. Atkinson, C., Kühne, T.: Rearchitecting the uml infrastructure. ACM Trans. Model. Comput. Simul., volume 12(4) (2002):290–321. ISSN 1049-3301. doi: <http://doi.acm.org/10.1145/643120.643123>
8. Engels, G., Hausmann, J. H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: A. Evans, S. Kent, B. Selic (eds.), UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings, volume 1939 of LNCS. Springer (2000), 323–337
9. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: A. Hartman, D. Kreische (eds.), Model Driven Architecture Foundations and Applications: First European Conference. Springer (2005), pp. 115 – 129. doi:10.1007/11581741
10. Muller, P.-A., Fleurey, F., Jzquel, J.-M.: Weaving executability into object-oriented meta-languages. In: L. Briand, C. Williams (eds.), Model Driven Engineering Languages and Systems: 8th International Conference, LNCS. Springer, Montego Bay, Jamaica (2005). ISBN 3-540-29010-9, pp. 264 – 278. doi:10.1007/11557432
11. Wagelaar, D., Jonckers, V.: Explicit platform models for mda. In: MoDELS (2005), 367–381
12. Scheidgen, M.: A MOF 2.0 for Java. URL <http://www.informatik.huberlin.de/sam/meta-tools/aMOF2.0forJava>. Last checked: February 8, 2006
13. Scheidgen, M.: CMOF-Model Semantics and Language Mapping for MOF 2.0 Implementations. In: R. J. Machado, J. ao M. Fernandes, M. Riebisch, B. Schätz (eds.), MBD/MOMPES. IEEE Computer Society (2006)