Operational Semantics for MOF Metamodels
Tutorial on M3Actions

Michael Soden,
soden@ikv.de

Department of Computer Science, Humboldt University Berlin
Unter den Linden 6, 10099 Berlin, Germany

Abstract. This tutorial is an introduction to the M3Actions: a framework to support the definition of operational semantics in MOF metamodels. We give an overview on the ingredients of the framework including the extended instantiation concept and the MAction language. The action semantics language is explained along with a small example metamodel to show how executable models can be defined and executed.

1 Introduction

Over the last years, new engineering paradigms of model-centric development or language-oriented programming drew the attention of software engineering communities. Evolving domain specific engineering manifested a general call for more efficient software engineering with customized modelling languages accompanied by first-class tooling. While the classic approach to mostly textual language development is rooted in the history of automata theory and parser technology, newer frameworks are driven by the concept of higher-level, object-oriented *metamodelling*. In this context, the M3Actions strives for providing a framework that supports the definition of executable language definitions based on MOF metamodels.

Reaching back to previous work at the Humboldt University Berlin on execution semantics of the SDL language using Abstract State Machines [1], the design rational of M3Actions addresses the need to have a framework for human readable, high-level, but precisely executable definitions of languages. Main aim is the formal analysis and assessment of executable models by means of simulation and testing. Its current implementation is based on top of the eclipse modeling projects EMF, MDT-OCL and GMF [2] and designed to be an extensible framework centered around an exectuable metamodelling core.

For the definition of models one can assign the different parts to either the structure (abstract syntax), static constraints, representation (concrete syntax) or behaviour (execution semantics) [3]. Within this language definition grid, the M3Actions framework focusses on structure and behaviour aspects by means of operational semantics [4] and consists of the following parts:

– A MOF-based metamodelling facility enhanced by explicit instantiation for the definition of structural aspects of models

– OCL for the definition of static constraints within models
– An action language to define the execution semantics by means of operational semantics.

The framework comprises an editor for metamodels (with extended instantiation relations), a graphical editor for the action flows and a (meta-)simulator to execute model instances with the behaviour defined in the metamodel. The simulator is realized as an interpreter that supports execution, debugging and execution trace recording for the analysis of executed models. To edit instances of specific metamodels under design, either the reflective EMF tree-editor or supplementary plugins for the concrete syntax might be used.

The remainder of this tutorial is structured as follows: section 2 explains the motivation of exectuable models and M3Actions architecture rationals. Afterwards section 3 gives a first idea how these concepts are applied using the famous 'Hello World' program, before section 4 dives into a more complex sample to exemplify the execution and instantiation concepts supported by the framework.

2 Executable Metamodels

The nature of model execution is not new: every program can be seen as an *executable model* if we consider the program's code to be *a model*. Hence, 'executable metamodels' or 'dynamic metamodeling' are actually misleading terms: not metamodels are executed, but their instances are. However, the aim of this terminology is to stress the definition of behaviour within a metamodel in contrast to the traditional way of defining execution semantics by a mapping into a so called *semantic domain*. Thus, M3Actions and related frameworks[1] support the definition of executable language definitions directly *inline* in the metamodel [10][11][12]. For this purpose, the M3Actions framework provides a visual action language called *MActions* which is similar to UML Activities/Actions to specify the operational semantics of models beside the pure structural object-oriented concepts known from MOF [13].

One of the key points in the architecture is to separate the fixed model structure as defined by ordinary metamodels from their changing runtime configurations: the *runtime model*. This separation might be artifical in the first place, but it helps to classify the different models involved and to distinguish between changing and non-changing parts. Figure 1 visualizes the (meta-)models involved in a language definition and their relationships. Beside the abstract syntax that is usually manipulated through an editor that visualizes the model in its concrete syntax, model execution might require additional runtime information to be managed, for instance program counters or active state descriptors, data structures such as stack frames, events, etc. The idea is to clearly separate these volatile objects from the fixed part of the model through *instanceOf*[2].

[1] e.g. XMF[5], Kermeta[6], GME[7], MetaEdit+[8], AMMA[9]

[2] Hence, we usually refer to the metaphor of *program* and *process*, where the latter is an instance of the former

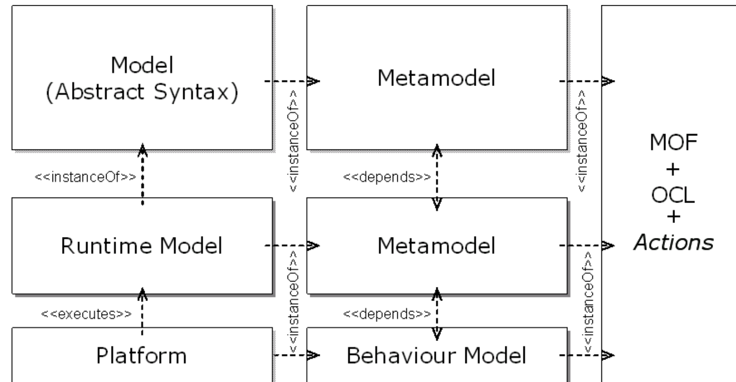

Fig. 1. Architecture overview and meta-layer relationships

Even though the MAction language supports the manipulation of model elements in general, we usually apply changes only to the runtime model[3]. Conceptually, we consider runtime models as being instances of the models of the abstract syntax, residing at 'level M0' of the classical meta-layer stack (cf. figure 1). For this purpose, the M3Actions framework supports an explicit *instanceOf* relationship as first-class modelling concept to distinguish between abstract syntax and runtime model(s).

2.1 Instantiation

Giving rise to the meta-muddle, one can define multiple logical meta-layers in an M3Actions metamodel. Our experience shows that one can draw advantages from such an explicit *instanceOf* relationship, especially when defining execution behaviour with runtime information that supplement the static abstract syntax. As consequence, however, we have to distinguish *physical* and *logical* instantiation in the sense of Atkinson et al[14]. We'll briefly discuss the concepts in the following.

Typically, today's object-oriented languages like Java, C# or C++ support *shallow instantiation*: a class can be instantiated directly and resulting objects are created in memory, receive slots for all (non-static) properties defined in the class, get initialized, etc. This type of instantiation is referred to as *physical* instantiation and normally strictly bound to the type-system of the language by means of introspection, inheritance and polymorphism. Programmers might check for type conformance through an 'instanceof' operator and (rather seldom) query for all instances of a certain type, because the language/execution envrionment manages this instanceOf relation.

The second type of *logical* instantiation is orthogonal to the former and usually not supported by ordinary programming languages. The semantics are

[3] any change to the abstract syntax would be rather an 'edit' operation, model transformation or a self-modifying execution

closer to an association that links together 'objects' and 'meta-objects'. Both instantiations share the conceptual circumstance that an object is *an instance of* another object. Nevertheless, while physical instantiation defines the structure (slots for values and links) of its instances, logical instantiation has no effect on the structure except that this additional 'object-level' instanceOf relation is managed by the metamodelling framework.

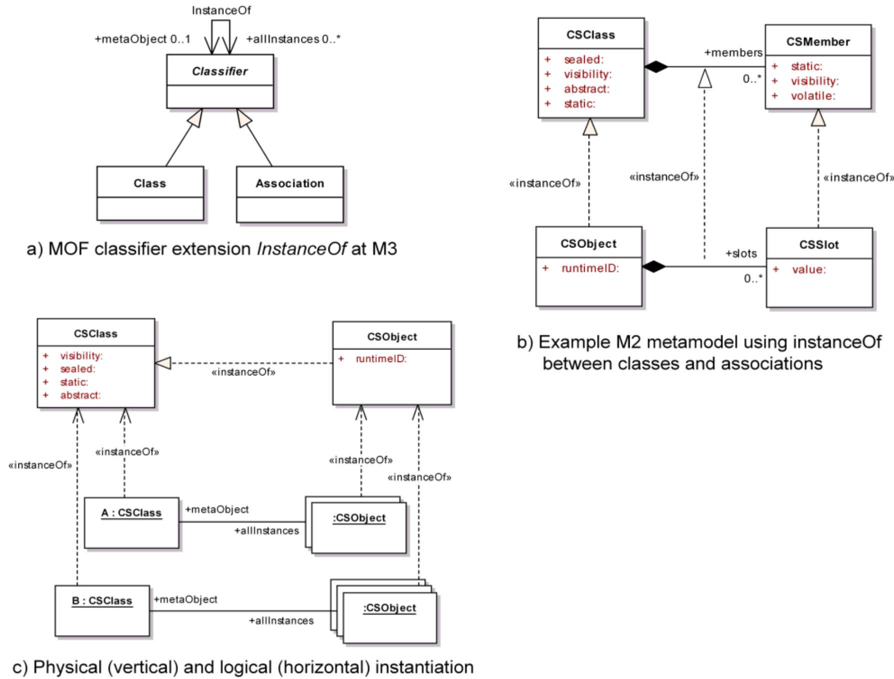

Fig. 2. InstanceOf extension and application

Figure 2 shows both concepts and their relation: a) shows the *instanceOf* association as extension of MOF at 'level M3'. This relation can be set between classes and associations[4] as shown in b). Part c) of the figure shows the relationship of physical instances and their logical *instanceOf*. The M3Actions framework manages this relation by injecting two references to every object: `metaObject` and `allInstances`. Conceptually, they're extensions to OCL reflection, technically they're added to every object during load and/or creation time of elements, respectively. There are two considerations to be taken into account to define the *instanceOf* semantics precisely: (1) how to distinguish between physical and logical instantiation and (2) how objects do receive their logical instanceOf relationship. For (1) the answer is depending on the context:

[4] Although Associations have not been implemented so far

the user has to know which instantiation he wants to check. Since in the overall framework OCL is used to navigate and access element properties, OCL must be capable of handling both instantiation properties. As one would probably expect, question (2) is answered at creation time of new objects. Before going further into the details of the creation of elements, we have to look at the actions that define the runtime manipulations and return to these questions during the example in section 4.

2.2 MAction language

Model elements are manipulated with a number of basic actions which make up the *MActions language*. The concrete (graphical) syntax as implemented is borrowed from UML Actions/Activities [15] using activity flows with pins, object nodes, etc. A purely textual syntax is currently not defined. In the same way as MOF does for the structural modelling, execution semantics of MActions are solely defined over MOF and OCL recursively (the later for expressions and types) by MActions itself[5] [13][16].

The MActions provide two specific extensions to specify model behaviour using actions: `MActivities` and `MOperations`. While the former define standalone behaviour flows without any context, the latter are hooked into classes as operations that have an action-based behaviour definition. Thus, operations have always a `self` variable defined, referring to the instance on which the operation was invoked. The native actions include the following:

MQueryAction executes an OCL query over the model and returns the result at the output pin. It is parameterized with arbitrary inputs that are available as local variables in the query expression

MAssignAction manipulates object properties by assigning/adding/removing values of properties (single and multi-valued)

MCreateAction Instantiates a specified meta-class and returns a new instance at the output pin. Optionally, it sets the logical instanceOf link

MInvocationAction invokes another `MOperation` on a given context object '`self`' or an `MActivity`. Additional black-box or library operations can be hooked in as extension

MIterateAction iterates a collection as result of an OCL query using an iterator that can be accessed from nested operations

MAtomicGroup groups a set of actions to build more complex behavior which is semantically *atomic*. Atomic behaviour is not interrupted in case of multiple threads

MInputAction Read or import a set of elements from the environment into the model and return it at the output pin. The execution framework is responsible of providing an adequate input as defined by the action's pin specification

[5] the recursive nature of the definition as kind of "meta-layer fixpoint" gave the framework its name

MOutputAction Output a set of model elements to the environment. The framework is responsible to process the output, e.g. print or visualize it, store it to a file, database, etc.

MDecisionNode Decision nodes are used to specify conditional flows. Conditions are expressed as disjoint OCL constraints (multiple outputs are supported).

Behaviour is always executed in the context of an `MThread`. Forking of `MThreads` is supported by the `MInvocationAction`. In order to describe the executable semantics of an MActions specification, we will explain the details along with the examples in section 3 and 4. Having defined the execution semantic of the models by means of a metamodel comprising abstract syntax, runtime model and actions, models can be executed by specifying the entry behaviour as input to the M3Actions interpreter.

3 Hello Meta-World!

Let's start with one of the simplest executable models: the Hello World program. Since we don't need any structure in our metamodel to just say hello to the world, we do not define any classes but only a standalone `MActivity`. In terms of UML, `MActivities` are classifier behaviours that build a unit of execution (i.e. they're not executed in the context of an object and have no `self` defined). For printing the string "Hello MetaWorld!", we need an action flow as show in figure 3.

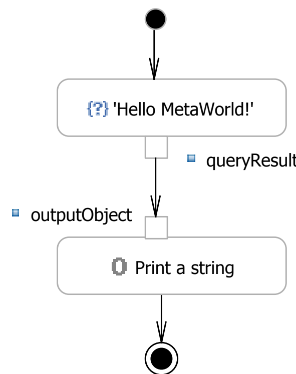

Fig. 3. Hello Meta-World as MAction behaviour

Any MAction flow has to start with an initial node and end at one (or more) final nodes. The first action in the hello world behaviour is an `MQuery` action. This action gets an OCL expression[6] and returns the result at the output pin

[6] the *Essentiel OCL* (EOCL) subset is supported (cp. [16])

(here: `queryResult`). The output pin is of type `String` and has a multiplicity of `[1..1]` (not shown by the visual editor). In this example, the OCL expression is a constant query and returns always the same string at the output pin. In case the pin specification would not match the query's result-type or multiplicity, the overall behaviour is undefined[7].

The second action is an `MOutput` action that delivers the string to the user. It has an input pin `outputObject : String` that specifies the expected objects to be shown to the environment. The format of presentation is not defined in the language, but by the executing environment (simulator framework)[8]. Even though we use a primitive as pin type, we could also pass *any model* as output. However, as one would expect from a programming language, the default is a simple (stringified) console output.

Before we continue with a larger example that uses structural concepts in the metamodel, we extend this small example by introducing the `MInput` action. Symmetrically to `MOutput`, this action has an output pin of a certain type that specifies the expected input objects. Let's say we want to read the name of the user in our example and produce the greeting: "Hello <*user-input*>", we could modify the behaviour as shown in figure 4. The new `MInput` action "Read a string" has a pin `name : String` at which the input from the environment is provided.

In the same way as for outputs, the input might be of primitive or complex type. If the environment does not deliver an input suitable to the pin specification, the behaviour is undefined. For example, if we use `Integer` as pin type, the environment must guarantee to provide a valid integer value in order to continue proper execution.

Beside the new input action, the query action has now a 'real' dynamic calculation of a concatenated string consisting of the value of variable `greet` and the result available at output pin `name`. Basically, OCL *let* statements defining variables and input pins are the same: both define variables accessible in the subsequent OCL expression. Note that the query will under no circumstances modify the objects passed in: it's truely side-effect free!

Furthermore, the gentle reader might have noticed the omission of the input and output pins of the query action. In general, it is necessary to specify pins for all passed inputs/outputs of an action. But if we look back to figure 3, we notice that both pins `queryResult` and `outputObject` are redundent, since they simply constitue a *renaming* of the reference to the object provided by the query. Hence, for simplicity reasons, consecutive pins which have the same name, type and multiplcity can be omitted. This kind of 'syntactic sugar' is intended for simplicity and to lower the burden of specifying redundent information.

---

[7] there is one exception to this rule: if the type matches and the output pin is of [0..1] multiplicity, but the query returns a collection, it is assumed that the collection shall be flattened to the first element

[8] The purpose is define a single environment interface that might be adjusted to specific needs for the models being defined

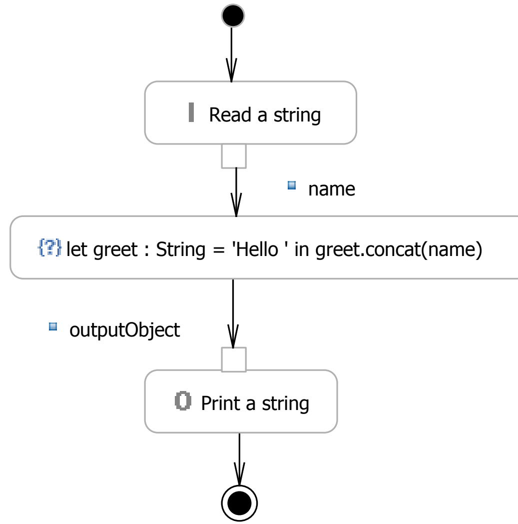

Fig. 4. Greeting with `MInput` action

4 Meta-layers and Instantiation

This section explains the execution semantics with their relation to the common structural class modelling concepts of MOF. As shown in the previous section, the elementary elements for behaviour modelling are object flows, basic action types and in a pivotal role: OCL. We introduce more advanced actions along with the small example metamodel that exemplifies *multiple meta-layers* (shown in figure 5).

The metamodel consists mainly of a container class `MultiLayerModel` and three classes which are in our focus with respect to execution: `MetaConcept`, `Concept` and `Instance`. As their name implies, each class defines a concept on a different *logical* meta-layer. That means, the metamodel describes entities of a model (MetaConcept), instances of that model (Concept) and instances of the instances (Instance). Even though this might seem odd at first glance, recall that for example the UML language has also constructs at (at least) two logical meta-layers: Classes and Objects[9]. For M3Actions, we realized the *explicit* modelling of this *instanceOf* relationship and support it in the action language and the framework (cp. figure 5). For a more detailed discussion on meta-layers and instantiation please refer to section 2.1.

The purpose of this three layer metamodel is to stress the difference between a fixed part of the model (usually referred to as the *abstract syntax*) and its *runtime model* changing over execution time. Typically, the abstract syntax part in the

[9] Note that the MOF standard itself speaks no longer from the strict 4-layer metamodel, but from objects and their meta-objects [**?**]

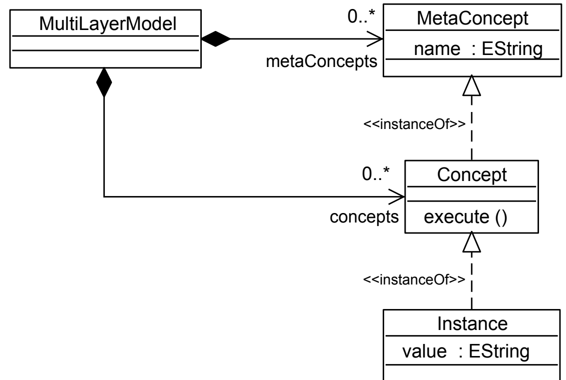

Fig. 5. Metamodel with three logical meta-layers: MetaConcept, Concept and Instance

metamodel is supported by textual or graphical tools to create, manipulate or transform these models. In the example metamodel, `MetaConcept` and `Concept` are used to represent the fixed part of the model, hence the containment relations to the `MultiLayerModel` class which serves as container for these objects[10]. Let's have a look at an instance of this part of the metamodel as shown in figure 6.

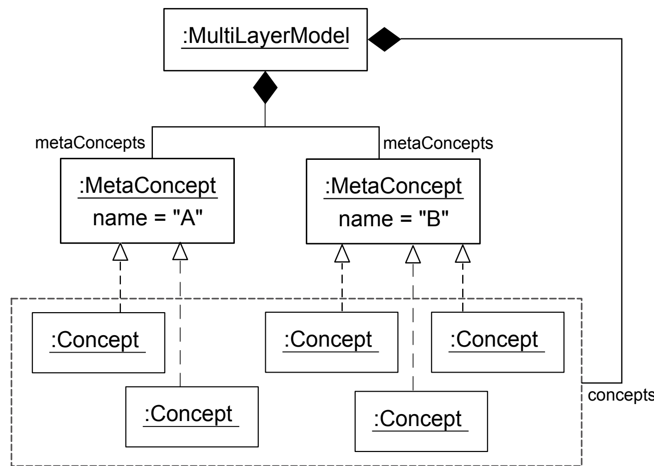

Fig. 6. Example instance of the multi-layer metamodel

This model is a *physical* instance of the metamodel depicted in UML object notation. Note that the objects are direct instances of the meta-classes and each

[10] actually EMP tooling usually requires such a top-level root container (e.g. GMF canvas, OCL extents for constraint checks, etc.)

object state its meta-class by name. Straightforward, properties such as `name` or `concepts` are slots or links, respectively. Important are the instanceOf relations: they are handled as links with special semantics (dashed lines with triangle). We'll see in a minute how they behave during execution. Note that this model does not include any instances of meta-class `Instance`: this part is supposed to serve as runtime model and we'll instantiate it dynamically during execution.

Now, let's define some behaviour in the metamodel to actually execute some behaviour over the models. First, we define a standalone MActivity `Startup` to initiate execution runs. The action flow is shown in figure 7. After the obligatory initial node, the central action is an `MIterate` action. Similar to predefined OCL iterators[11], the action provides an iterator variable that is assigned to one value of a collection at a time while executing a set of nested actions. In the example, the expression `concept in MetaLayers::Concept.allInstances()` defines the interator `concept` over the result of the `allInstances()` query. Note that the OCL `allInstances` operation works over the *physical* instanceOf relations, not logical ones. For the model in figure 6, this iteration would iterate over the five instances depicted of class `Concept`.

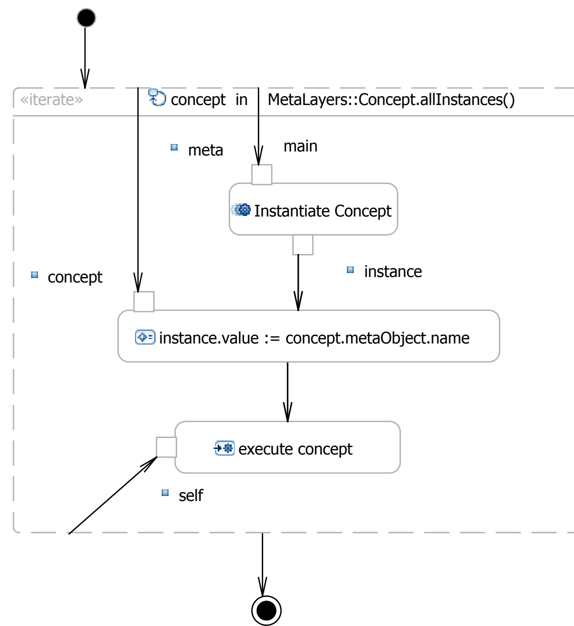

Fig. 7. `MActivity` *Startup*

[11] cp. [16], section 11.8

The first action *Instantiante Concept* inside the iteration is an `MCreate` action. In the first place, creation of objects behaves like in other OO languages, i.e. an object is allocated with slots for all properties of the given class to be instantiated. The (meta-)class to be *physically* instantiated is given by the output pin's type. Additionally, as the object's class might have an explicit instanceOf relation being defined, this *logical instanceOf* link can be set to such an `metaObject` during creation time by passing a conforming object via an input pin. Here 'conforming' means that the type of the output pin α and the type of the input pin β must have either a direct *instanceOf* relation, or one of the (direct or indirect) supertypes of α must have this relation to β. In our example metamodel, the input pin of the create action references at runtime the current iterator object which is of type `Concept`. Then it will produce an object of type `Instance` and create the logical instanceOf link between the two. As result, the newly created instance remains a logical instance for the whole object lifetime in the further execution and exposes this fact via its `metaObject` reference which can be navigated later on.

Next in the flow is an assign action which makes use of the `metaObject` property of the iterator `concept`. Naturally, the instanceOf link can also be set at the fixed models (in the editor) and not established dynamically at runtime as in the previous create action. Recall from figure 6 that each instance of class `Concept` really has this fixed logical instanceOf link set to an instance of class `MetaConcept` as shown. Now, we can navigate this link and access the `MetaObject#name` attribute of the iterator `concept` simply by writing: `concept.metaObject.name` and assign this value to the attribute `value` of the newly created object `instance`. Note that we're effectively dealing with all three (logical) meta-layers in this action: `MetaConcept` objects (level 1) are accessed by the `metaObject` reference from the iterator `concept` (level 2) and the query result is assigned to objects of type `Instance` (level 3).

The third action in the overall iteration is finally an `MInvocation` action. Invocations behave like normal OO method calls including polymorphism (virtual method dispatching, late binding). Not shown in the visual editor is the invocation target, which is `Concept#execute`. While the current `Startup` flow is an `MActivity`, the invoked operation is an `MOperation` (subclass of MOF `Operation`). This is the second kind of behavioural feature in MActions to define object level behaviours. Generally, the context of a behaviour is defined by a pin with the predefined name `self`. `Self` can be accessed within the flow from any action. Therefore, we have to specify a `self` object as context of the invocation: we use the iterator `concept` (see input pin in figure 7). If the invoked operation would declare any in/out parameters, for each parameter an input or output pin have to be present, respectively. These parameters would be accesible through object nodes in the invoked operation.

The behaviour flow of `execute` is depicted in figure 8. The behaviour consists of a query and a subsequent output. The main purpose of this behaviour is to demonstrate the handling of the previously created `metaObject` and its opposite direction: the *logical allInstances*. While the (single) meta-object of an instanceOf

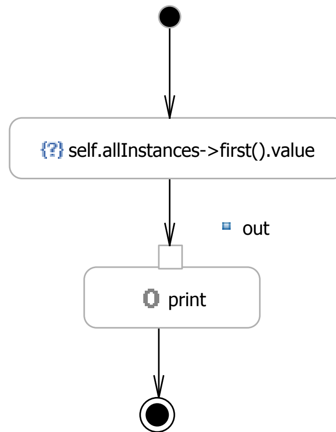

Fig. 8. `MOperation` *execute*

can be accessed by `metaObject`, all logical instances of an object can be retrieved by the `allInstances` reference. In the OCL query, the current context object is accessed through the `self` keyword, followed by a navigation to the first element of the logical `allInstances` collection. Note that while the physical instanceOf relation can be accessed with the OCL operation `allInstances()` at the class (e.g. as in the iteration in figure 7), all logical instances are accessed via this reference that is conceptually available as OCL reflective property at any object which class has the explicit instanceOf defined. Hence, the result of this query will be the previously created `Instance` object which carries the `name` of its (logical) meta-meta-object[12]. Therefore, the output action prints the value A or B for each 'instance-instance' of `MetaConcept` objects which have the respective name A or B.

4.1 Acknowledgements

We would like to thank our colleagues at the Humboldt University Berlin for ongoing support and review. The contents of this tutorial is part of the author's collaborative work with Hajo Eichler on their Ph.D. thesis.

References

1. Prinz, A.: Formal Semantics for RSDL: Definition and Implementation. PhD thesis, Humboldt-Universitt zu Berlin (2000)
2. Eclipse Project: (Eclipse Modeling Project (EMP))

[12] For simplicity, we create in *Startup* only one instance and use the `first` collection to retrieve it

3. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based language development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (2004)
4. Plotkin, G.: A structural approach to operational semantics. Technical report, University of Aarhus, Denmark (1981)
5. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodeling, A Foundation for Language Driven Development. Xactium (2004)
6. Team, T.: (Triskell Meta-Modelling Kernel. IRISA, INRIA. www.kermeta.org.)
7. 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
8. MetaCase: (MetaEdit+. http://www.metacase.com.)
9. Davide Di Ruscio, Frederic Jouault, I.K.J.B.A.P.: Extending amma for supporting dynamic semantics specifications of dsls. Technical report, Universite Studi dell'Aquila (2006)
10. Soden, M., Eichler, H.: An approach to use executable models for testing. In: Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures. Volume P-119 of LNI., GI (2007)
11. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. (In: ECMDA)
12. Eichler, H., Soden, M., Scheidgen, M.: A semantic meta-modelling framework with simulation and test integration. In: ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing, Bilbao. (2006)
13. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group (2003) ptc/03-10-04.
14. 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
15. OMG: UML 2.0 Superstructure Specification. Object Management Group (2004) ptc/04-10-02.
16. OMG: OCL 2.0 Specification. Object Management Group (2005) ptc/2005-06-06.