Object-Oriented Operational Semantics

Andreas Prinz[1(✉)], Birger Møller-Pedersen[2], and Joachim Fischer[3]

[1] Department of ICT, University of Agder, Grimstad, Norway
andreas.prinz@uia.no
[2] Department of Informatics, University of Oslo, Oslo, Norway
birger@ifi.uio.no
[3] Department of Computer Science, Humboldt University, Berlin, Germany
fischer@informatik.hu-berlin.de

Abstract. Operational semantics is one way of providing meaning to an executable language. On a high level of abstraction, operational semantics means to define an interpreter or an abstract machine for the language. In this article, we review the concept of operational semantics in the scope of meta-model-based language definitions and identify challenges and issues. We provide a clean conceptual approach using an object-oriented runtime environment and state change operations, which relies on an underlying abstract virtual machine. We present the approach using a sample language.

1 Introduction

Modelling and meta-modelling are important approaches in the scope of OMG's MDA framework [8]. The table below shows the OMG four level architecture and the corresponding concepts for grammar-based definition of languages.

OMG level	Examples	Grammar example	OCL example
M3: meta-languages	MOF	EBNF	MOF
M2: languages	UML metamodel	Java grammar	OCL language
M1: models	UML model	a program	a formula
M0: instances	runtime objects	a run	a truth value

MDA is mostly concerned with models and programs, which are placed on OMG level M1. Extending MDA to domain-specific languages lifts the focus from M1 to M2, where languages are described.

We start with the understanding that a language on M2 is the collection of instances of the metaclasses that define the language together with their semantics. An instance of a programming language is called a program, and an instance of a modelling language is called a model. In this paper, we use the term language instance for both of them.

When (formally) describing languages on M2, one would typically consider the three aspects structure, presentation, and semantics. A language *structure* defines the set of all possible language instances and restrictions on this set.

J. Grabowski and S. Herbold (Eds.): SAM 2016, LNCS 9959, pp. 132–147, 2016.
DOI: 10.1007/978-3-319-46613-2_9

It consists of metaclasses for its concepts. The *presentation* aspect can include descriptions of textual, graphical, and tabular presentations and a mixture of these. The *semantics* aspect describes the meaning of language instances. It can be given using different methods, e.g. describing language instance execution (execution or operational semantics), mapping into another language (transformation semantics), or defining a mathematical relation between input domain and output domain (denotational semantics).

This paper focuses on structural operational semantics (SOS), which is a way of directly defining how language constructs are executed by providing execution sequences (traces of configurations) as the semantics of a program. The traces are given by a set of inference rules, describing how state changes happen.

Let us consider configurations with three parts $< i, v, s >$, with a language instance i, a current value v, and a storage s. To describe the semantics of a statement **skip**, we use the following rule, which states that **skip** can be replaced by the empty language instance ($\perp$). The arrow denotes the state change relation.

$$\frac{}{< \textbf{skip}, v, s > \rightarrow < \perp, v, s >}$$

SOS also allows introducing steps on different levels as shown with a sample inference rule handling an assignment of a location L with an expression E. The precondition of the rule is that the expression E is reduced with several steps ($\rightarrow^*$) to empty ($\perp$), thereby producing the value v' and the state s' (i.e. E evaluates to v'). The rule itself turns the assignment statement into a state update, where the value for the location L is changed to v'. So the assignment has one step of updating L, but underneath there are several intermediate steps of evaluating E. In this case we use the transitive closure of the steps ($\rightarrow^*$) instead of a single step ($\rightarrow$).

$$\frac{< E, v, s > \rightarrow^* < \perp, v', s' >}{< L := E, v, s > \rightarrow < \perp, v', s' \uplus \{L \mapsto v'\} >}$$

Defining SOS for **while** statements is a bit more involved, as the following example shows. The first rule tells us that the while statement does nothing in case the condition B evaluates to **false** in the current state. The second rule handles the case where B evaluates to **true**. In this case, we have to execute the body S followed by the complete loop.

$$\frac{< B, v, s > \rightarrow^* < \perp, false, s' >}{< \textbf{while } B \textbf{ do } S, v, s > \rightarrow < \perp, v, s' >}$$

$$\frac{< B, v, s > \rightarrow * < \perp, true, s' >}{< \textbf{while } B \textbf{ do } S, v, s > \rightarrow < S; \textbf{ while } B \textbf{ do } S, v, s' >}$$

This last example shows the roots of SOS in functional programming, where it is a normal idea that programs can change at runtime. For non-functional programming languages this style might not be as appealing. For object-oriented languages in particular, there are some inconveniences with this style of language definition.

1. In the programming world the program itself is typically considered to be fixed. As the last example shows, the language instances in the configurations are changed in order to indicate the position of execution. For imperative programming, one would rather think of a program counter to indicate the current position.
2. The state of execution is only partly considered. For object-oriented programming, a rich object-structure would be envisioned.
3. Typical runtime structures as program counters and call stack are not directly visible in SOS.

This paper tries to provide a way to describe operational semantics for an object-oriented situation. The paper is conceptual, and uses an example to explain the approach. It does not propose a concrete tool or language.

We continue this paper in Sect. 2 with a discussion of operational semantics and introduce the sample language in Sect. 3. We look into configurations in Sect. 4, and the relation of operational semantics to an execution platform in Sect. 5. After discussing execution semantics in Sect. 6, we summarize in Sect. 7.

2 Operational Semantics Description

There are several forms of operational semantics, in particular structural operational semantics (SOS - also called small-step semantics) [12] and natural semantics (also called big-step semantics) [7]. The first class focuses on the individual computation steps, while the second has more focus on how the computation results come about. For our purpose these differences are marginal and we use SOS as a reference. Operational semantics use the understanding that computations are sequences of runtime states, where the states are called configurations (see e.g. [4,7,10,12]). An SOS transition system is given by a set of configurations, a set of labels and labelled transitions, as well as a set of initial states and a set of final states. The typical idea of an SOS is to change configurations. Control information is often encoded by changes to the current language instance.

Similar elements are defined in other approaches for semantics definition. Rewrite-based approaches as K [13] and Maude [2] relate to configurations with the main aim to reduce the computation to its final value. Rascal [9] provides similar support as SOS, but is more code-oriented as also our approach.

For our definition of operational semantics, we use similar elements. We assume a structure aspect description of the language L. The structure could be defined using MOF (EMF) [3], but any other suitable language would work. Starting from the structure of L, we need to define two parts of semantics. First, we define the *structural part* of the runtime of L, i.e. its set of runtime states (configurations)[1], which we call runtime environment ([15]). Second, we have to handle the *dynamic part* (execution in terms of state changes).

[1] Please note the difference between structure of the language (its constructs, e.g. if-construct), and structure of the runtime (its runtime elements, e.g. a stack frame).

Commonly, the *structural part* is called instantiation semantics or structure-only semantics [15]. It defines how the elements of the language relate to runtime elements that can be created and used at runtime. For classes, this would typically be objects, while for methods, we envision stack frames. Depending on the language, also message buffers, or an exception stack could appear. In traditional operational semantics approaches, the main focus is on state changes, while the configurations are defined ad-hoc. However, in our object-oriented approach, configurations are very important.

The *dynamic part* describes the actual state changes that take place at runtime. All execution sequences are based on trivial changes, which are implied by the structural part. These trivial changes are creating new objects, setting values, and adding objects to collections. The dynamic part describes how to combine the trivial changes into bigger changes, in many cases a complete run.

In the operational semantics world, the main focus is on describing languages with a completely defined runtime behaviour, i.e. in any state the possible next states are determined. For example, this would be the case for Java. In practical terms, it means that the program runs to completion.

In interpreted languages and script languages, the combined trivial changes are not complete runs, but runs that can be put together by a user to even bigger runs. An extreme is a language like MOF, which only defines trivial state changes, which can be combined by a user using an editor. We discuss in Sect. 6 how state changes defined by a program are sequences and combinations of these trivial changes.

The main contribution of this paper is the handling of the structural part in Sect. 4, and its relation to the underlying machine and the structural part of this machine as discussed in Sect. 5. Operational semantics normally considers this underlying machine as given, and is not aware of its structure and behaviour. In our approach, the language designer may choose the features of the underlying machine and design the operational semantics combining trivial behaviour and underlying behaviour.

3 SLS - A Sample Language

In order to illustrate and discuss our approach, we consider SLS, a simplified version of SLX [6], an executable language for simulation of dynamic systems. SLS is based on principles of next-event-progressions. Models of existing or hypothetical systems are built in SLS by describing their components as *objects* of classes. For each *class* of objects, attributes and methods are defined describing the structure of the identified system components and their behavior.

SLS distinguishes active and passive classes (and objects). Passive objects are objects that can only be acted upon. Active objects have a *main*() method describing the sequence of executable statements by which they operate on their own. The behaviors of all existing active objects including the system specification determine at runtime the complete behavior of the system.

A *main*() method can only be called indirectly, by activating the corresponding active object from another active object: **activate** obj_reference.

If the *main*() executes a **wait**, the further execution of the active object is frozen. It can be continued by another object: **reactivate** obj_reference.

SLS features global dimensionless model time, i.e. time that is controlled by the execution of the system, and that is not external to the execution. This allows to mimic a real system by having multiple activities carried out at the same time. An active object can experience scheduled delays in model time using the **advance** construct. In SLS, model time is viewed as a succession of instants. At any given instant in model time, the runtime system of SLS processes one by one all events that take place at that time. After that, the runtime system advances the model time to the next imminent event time.

For each active object, the runtime system needs to keep track of at least three things: (1) the location of the next statement to be executed for the activity in the program, (2) the location of data local to that object, and (3) the model time at which the object is to resume (in case of scheduled delays).

In Fig. 1, we show a simple SLS example. Our system consists of two factories (objects of class *Factory*) who produce products (objects of class *Product*) with an individual production time. Both factories run in parallel and deposit their products in a common collection (*ProductList*). As soon as 1000 products are produced, the simulation run should stop with a printout.

The *Factory* is an active class and the *Product* is passive. The *productList* is also a passive object (essentially a set of *Products*). Both the current number of deposited products and the maximum number have been defined as attributes of the global system (active) class *Production*.

The *dynamic semantics* of SLS is a simulation of the system, sometimes called simulation semantics. The *execution* of the SLS sample system starts with an instance of the system specification (here *Production*). The actual run is started by calling the *main*() method of *Production*. At runtime, each active

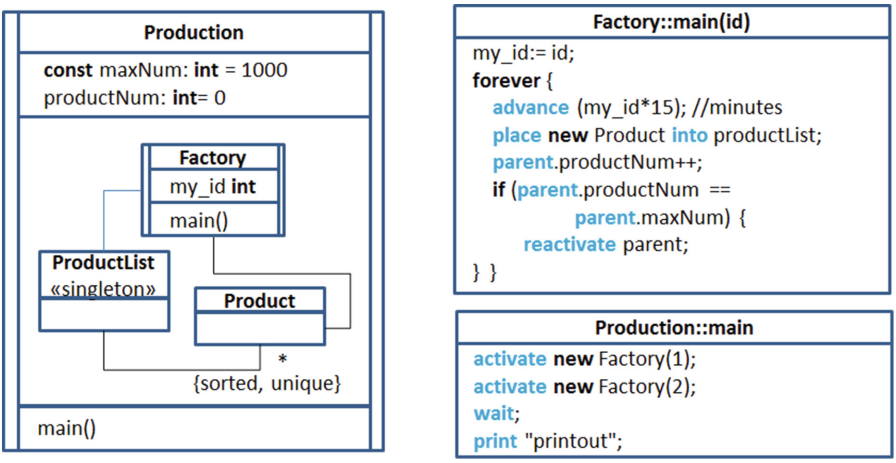

Fig. 1. Sample SLS specification

class (including the system specification) object has its own thread of control as well as a state. The following states are possible.

State	Semantics	Sorting principle
moving	The object is in the *moving* list. It is active or waits for execution. The time of the object is the current model time.	position in *moving* list
scheduled	The object is in the scheduled list. It is delayed by a defined time.	time of next move
waiting	The object is in the waiting list. It is suspended for unknown model time.	FIFO
terminated	*main*() of the object is finished; object cannot be reactivated	none

Now we look at the simulation run of our example, see the table below. The *Production* object is in the *moving* list, its time is 0 and the model time is 0. Its *main*() is called and generates two active *Factory* objects. Both of them get move time 0, and are placed in the *moving* list. Then *Production* executes a **wait** and is transferred into the *waiting* list, keeping its current execution state.

Time	Who	What	Delay until
0	Production	Create Factory(1) and Factory(2)	Infinite
0	Factory (1)	Delay 15	15
0	Factory (2)	Delay 30	30
15	Factory(1)	Create product-1	30
30	Factory(2)	Create product-2	60
30	Factory(1)	Create product-3	45
45	Factory(1)	Create product-4	60
...			
9975	Factory(1)	Create product-997	9990
9990	Factory(2)	Create product-998	10020
9990	Factory(1)	Create product-999	10005
10005	Factory(1)	Create product-1000, resume Production	10020
10005	Production	Finish simulation run	stop

The control now turns to the first entry of the *moving* list, the *Factory*(1) object. Its *main*() just delays for 15 units (we consider this to be minutes). Doing so, the *Factory*(1) will be placed into the *scheduled* list with move time 15. A similar action is done for *Factory*(2) with a move time of 30.

Because the *moving* list has become empty, it will be filled with all objects in the *scheduled* list with minimum move time. This also advances the model time to that time. Here, only *Factory*(1) is at time 15. It generates a new *Product* object, places it in the global *productList*, and delays for 15 more units. The further processing is indicated in the table. We show the runtime situation at system time 45 in Fig. 2.

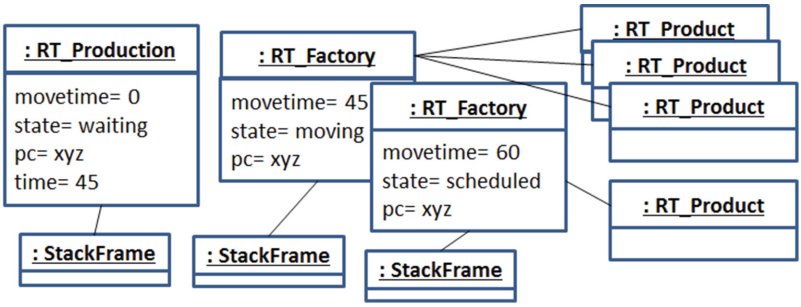

Fig. 2. Runtime situation at system time 45

Finally, the maximum is reached and the waiting *Production* is reactivated. After printing some information, the *main*() of *Production* finishes and this stops the simulation, because *Production* is the system instance.

4 Runtime Environment: Configurations

The runtime environment (RTE) is the set of possible states at runtime, i.e. configurations. Runtime states are purely structural and the state changes (Sect. 6) are based on them, see also [14]. The RTE depends on the language, i.e. the RTE description has to be done on level M2. In addition, the RTE can depend on the specific language instance, which has to be handled on level M1. We want to distinguish several kinds of RTE elements.

- The *read-only program* is included in the RTE, such that the execution can refer to the program.
- *Global elements* are only dependent on the language, e.g. predefined libraries, and program counter. They are independent of the specific program.
- *Local elements* are runtime elements that relate to language concepts and describe how these are instantiated. They are related to their respective concepts, but are independent of the language instance. There are three main cases as follows.
  1. *None-elements*, which means that the language concept does not have a runtime representation. An example would be statements and constant declarations in SLS. The relation from concept to runtime is 1:0.
  2. *One-elements* are extensions of concepts in terms of a 1:1 relationship. An example are locations for global variables in SLS, or the instance of the SLS system specification.
  3. *Many-elements* are also related to concepts, but with the possibility of many instances of the same element. A property of a passive SLS class is an example - it exists for each instance of the class. A stack frame for a method is another example, which exists for each call of the method. They have a 1:n relation from concept to runtime.

– *Dependent elements* are similar to local elements, but they depend on the specific language instance. Active and passive classes in SLS are examples here. They cannot be defined statically on level M2. Instead, on M2 a mapping from the language concepts to the runtime structures can be defined.

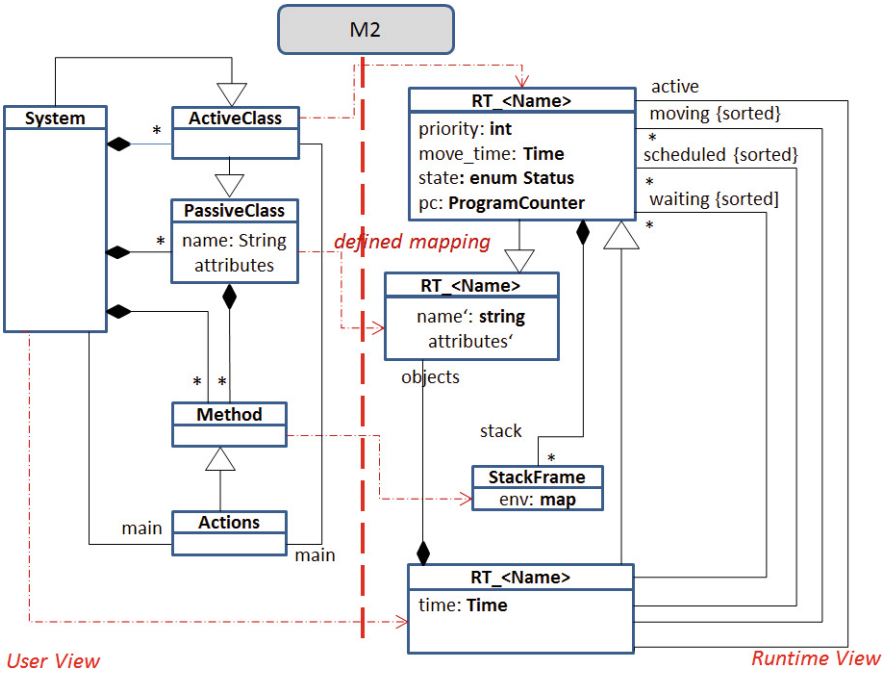

Fig. 3. Structure and RTE for SLS

When defining the RTE of SLS, we start at level M2. As the RTE defines instantiation semantics, it defines the transition from M1 to M0. We assume an underlying mechanism that can instantiate objects from classes, which is called MOF-VM. It is explained in more detail in the next section. For the RTE we describe which MOF-VM classes to use on M1, which implies the possible MOF-VM objects on M0. The crossing of the level boundary between M1 and M0 is done by the MOF-VM semantics. At level M2 it is not possible to define M1 classes. Therefore, at M2 we define a mapping from language concepts to RTE (see Fig. 3). This mapping is applied for the actual language instance, which yields the RTE at level M1 (see Fig. 4). Being a mapping, the RTE is defined on M2, and still depends on the language instance on M1. In Fig. 3, the *dependent elements* are active and passive classes as well as the system. The dependency in all cases are the available attributes. Please note that the mapping for SLS classes is not as trivial as for UML classes, as we need to take care of concurrency by including program counter (pc), call stack and time in the classes. For methods, we define a *local element* mapping to stack frames.

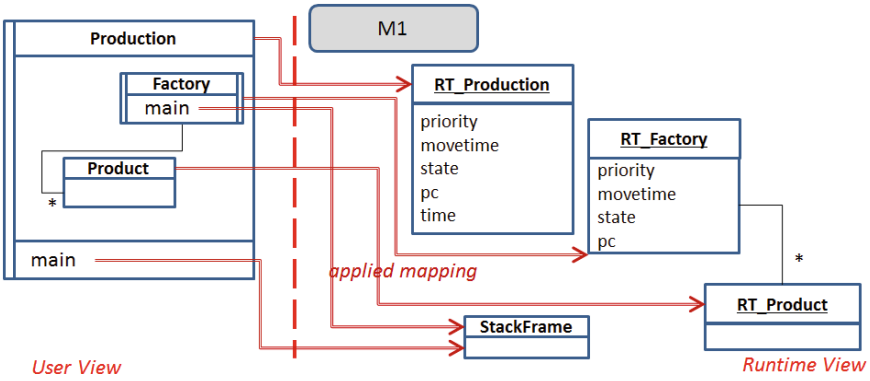

Fig. 4. RTE for the sample language instance

Figure 4 shows how the RTE mapping is applied to our sample language instance at level M1. The RTE mapping defined at M2 is now applied, and yields the MOF-VM classes *RT_Factory*, *RT_Product*, *RT_Production*, and *StackFrame*. Using this RTE, it is straightforward to instantiate the MOF-VM classes, yielding the situation at time 45 at level M0 as shown in Fig. 2.

This way, the RTE is defined at M2 as a mapping from the language instance on M1 to MOF-VM classes on M1, which then get instantiated at M0. Any mapping notation can be used, for example QVT [11]. In this article, we use an ad-hoc notation in order to explain the main idea.

The same approach is used for handling initial states and final states. All these states have to be formed according to the RTE. They are also given by mappings that are defined on level M2, and applied on level M1. Finally, initial states and final states are used on level M0.

5 Execution Platform: MOF-VM

Instantiation semantics is based on an underlying mechanism that provides basic instantiation. This could be very low level as in machine code, where an indication of a memory area leads to the provision of actual memory, thereby providing very simple instantiation. It could be more high level like a Java virtual machine, where a class instantiation mechanism is available. We assume a machine that can instantiate objects from classes, which we call MOF-VM ([5]). MOF-VM instantiation is the only way to do instantiation, such that all instances existing are MOF-VM objects, including the objects on M2, M1, and M0.

In Fig. 5 we show how the language definition and the instantiation semantics work together for the semantics of the M3 language MOF. The concepts of SLS are defined on level M2 as instances of the *Class* concept of MOF (which is defined on level M3), e.g. the class *ActiveClass* on the top left. This class has a MOF-VM RTE as defined for MOF, which is shown on the top right part of Fig. 5. Remember that MOF-VM is independent of the levels and provides the

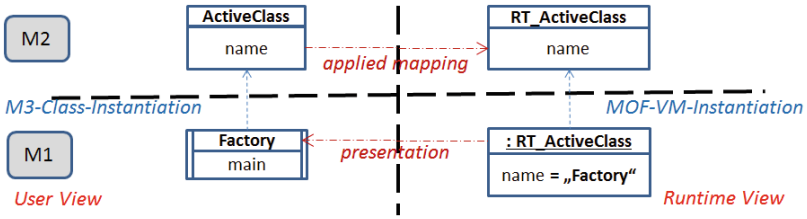

Fig. 5. MOF-VM instantiation versus language instantiation.

general way of crossing from one level to the one below. Using *RT_ActiveClass*, it is possible to create instances using MOF-VM semantics, and provide an object with name *Factory* (bottom right). This object can be presented using the custom presentation for *ActiveClass* (within SLS). This leads to the presentation on the bottom left of Fig. 5.

On the left side, *Factory* is a MOF instance of *ActiveClass*. This language-defined instantiation is based on the MOF-VM instantiation (right side) via the applied mapping and the presentation.

Going one level up, the applied mapping is based on the defined mapping on M3 (Fig. 6). There will be all of the three *presentation*, *defined* and *applied* mappings on the levels M2 and M3; for better understandability, we have only shown the relevant ones.

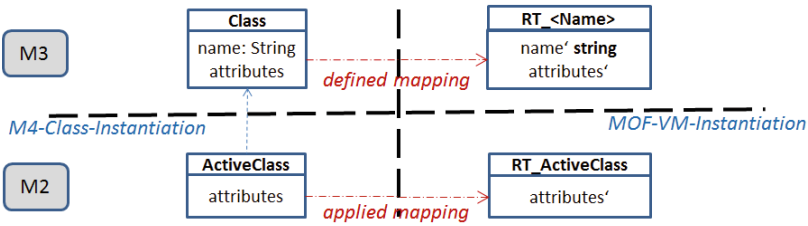

Fig. 6. Defined mapping versus applied mapping.

It is important to be aware that the underlying machine itself has already some built-in runtime structure that may or may not be used by the definition of the operational semantics. When we look at SOS as a language itself, the SOS-VM is functional[2], which means we expect it to be able to keep local values, and to handle (recursive) functions (see also Sect. 6). Look at the following SOS rule for a sequence construct to understand what that could imply.

[2] SOS as a language (on M3) is functional, not the language described using SOS (on M2). Of course, SOS can be used to describe all kinds of languages.

$$\frac{< S_1, v_0, s_0 > \rightarrow * < \bot, v_1, s_1 > \quad < S_2, v_1, s_1 > \rightarrow * < \bot, v_2, s_2 >}{< S_1; S_2, v_0, s_0 > \rightarrow < \bot, v_2, s_2 >}$$

The precondition of the rule is that a statement S_1 is evaluated to v_1, and S_2 to v_2. Then the sequence of S_1 and S_2 is evaluated to v_2 in one step. The steps inside S_1 and S_2 are intermediate steps in this case. In this rule, it is not clear where the value v_2 is stored – it is implicit in the underlying machine. This is possible, because the SOS-VM allows to transfer the value from the precondition (intermediate steps: above the line) to the top-level step (below the line) by just giving its name. This way the local intermediate value is kept in the RTE of the SOS-VM, and not in the RTE of the language. Similarly, the evaluation of a statement can imply stack frames to be created. This is not explicit here, because the SOS-VM facilities are used. Again, this means this part of the RTE is hidden by using the SOS-VM.

The language developer has to decide where to place the RTE elements like stack frames and local intermediate values. Either they are visible in the defined RTE, or they are hidden and implicit by using the SOS-VM or MOF-VM.

6 Run Versus Step: Dynamic Semantics

As discussed in Sect. 2, an operational semantics has to describe configurations, initial and final states, as well as state changes. In this section, we discuss state changes based on the configurations defined in Sect. 4.

The RTE object structure has an implied navigation along its links, including the navigation from objects to the language instance itself, and back. We assume that the navigation is rich enough to also include basic elementary functions of the underlying basic data types (MOF-VM data types), like boolean, integer, and collection operations. This way, the navigation is a query facility allowing to extract values in a very general sense (r-values). Furthermore, the navigation allows to extract locations, where it is possible to change the state (l-values). Simple updates are also implied by the RTE, and form the trivial state changes.

Defining executions is the combination of these trivial state changes into larger changes. It is the language that defines the granularity of the state changes.

For the object-oriented version, we use an adaptation of abstract state machines (ASM) [1] as the meta-language to define state changes. ASM normally come with an own underlying sub-language for the description of locations and values, but here this is implied by the RTE as follows.

- The RTE provides expressions (navigation/queries) and locations including the notation **new**(C) to create a new element of a MOF-VM class C.
- An RTE update has the following syntax. `<location> := <expression>`

On top of these trivial updates, we use the following ASM constructs for grouping of updates. As the semantics of the constructs is obvious, we do not describe it here and refer the reader to [1] in case of doubt. In ASM, an instruction is called *rule*.

- Decision instruction: `if <expression> then <rule> [ else <rule> ]`
- Parallel execution: `<rule> <rule>`
- Sequential execution: `<rule> seq <rule>`
- Named rule: `<name>(<name-list>) { <rule> }`
- Calls of named rules: `<name>(<expression-list>)`
- Local names: `let <name> = <value> in <rule>`
- The ASM constructs **forall**, **extend**, and **choose** are not used here.

ASM normally have a global view on state changes, and would define the semantics as a collection of global rules. As we are interested in an object-oriented approach, we attach the rules to the appropriate classes. These classes can be RTE classes or metamodel classes. As an example for RTE classes, we show methods of the *RT_Factory* runtime class. Essentially, the system (*RT_Factory*) delegates the handling of program counter and stack to the currently active object. Please compare with the runtime structure in Fig. 3.

```
RT_Factory::setPC(newPC) { active.pc:= newPC }
RT_Factory::setValue(idx,v) { active.stack.top.env.add(idx,v) }
RT_Factory::getValue(i) { return active.stack.top.env.value(i) }
RT_Factory::push(stackframe) { active.stack.push(stackframe) }
RT_Factory::pop() { active.stack.pop() }
```

We explain our approach with the three SLS statements shown in Fig. 7.

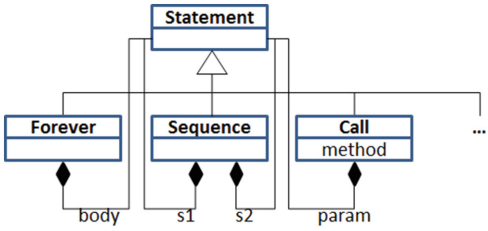

Fig. 7. Some SLS statements shown as part of the SLS metamodel

In SOS, this abstract syntax would be formalized as follows, where S are instances of statements, and M is a method name.

```
statement ::= forever S | S1;S2 | call M(S)
```

There are two possible ways of defining state changes in our approach, which we call STEP and RUN. Doing the STEP variant means to specify the steps of the language soleily based on changes in the RTE without using MOF-VM RTE. The RUN approach is to specify the whole execution including all the steps, possibly making use of the MOF-VM RTE. It is often easier to define the RUN approach, but it will hide part of the RTE as it is kept in the underlying MOF-VM RTE. We explain the operational semantics using the STEP and the

RUN approach by defining methods *step*() and *run*(), and contrast it with the SOS version. Please note, that STEP and RUN are alternatives, and in a real case the language developer would select one of the two (and not both).

The semantics of the **forever** statement has to handle two situations. (1) When entering a forever loop, the first step is to enter the body. (2) After finishing the body, the forever has to be re-entered again.

The SOS semantics of it is given with a rule that just calls the *body* and then calls the **forever** again. Thus, in SOS both situations are handled by a change to the program code.

$$\frac{}{< \textbf{forever}\ body, v, s > \rightarrow < body; \textbf{forever}\ body, v, s >}$$

The definition using RUN is very similar to SOS, the main difference being the attachment of the rule to the syntax class. Both situations are handled by calls to the *run*() method. A sequencing of the underlying machine (**seq**) is used to separate these two (big) steps.

```
Forever::run() { body.run() seq self.run() }
```

In STEP, the current position in the execution has to be handled explicitly. Situation (1) is covered by the *step*() routine of class *Forever*, setting the program counter. Situation (2) is covered when the inner statement is finished. In this case, a continuation has to be provided by its parent, which is done with the *nextPC*() method of class *Forever*. The definition of the semantics of the sequence construct shows the use of the *nextPC*(). We use *RTroot* in order to refer to the root of the runtime environment of type *RT_Factory*. This example shown nicely how the code of the syntax class *Forever* interacts with the code of the RTE class *RT_Factory*.

```
Forever::step() { RTroot.setPC(body) }
Forever::nextPC(node) { return self; }
```

The semantics of the *SLS sequence statement* includes three situations to handle. (1) When entering the sequence, we start with the first statement. (2) When the first statements is finished, we continue with the second statement. (3) When the second statement is finished, the result of the sequence is the result of the second statement.

In SOS, it is given by the following two tules. The first rule handles the case where the first statement can be reduced to an empty statement starting from a state s. In this case, it is sufficient to start the second statement from this new state, implementing the second situation. If this is not possible, the second rule describes that the first step in the execution of the sequence is the first step of the first statement, which implements the first situation. The second statement is unchanged in this case. The third situation does not need an extra handling in this specification, as the value of the second statement is automatically the final value. A different formulation is discussed in Sect. 5.

$$\frac{< s1, v, s > \rightarrow < \bot, v', s' >}{< s1; s2, v, s > \rightarrow < s2, v', s' >} \qquad \frac{< s1, v, s > \rightarrow < s1', v', s' >}{< s1; s2, v, s > \rightarrow < s1'; s2, v', s' >}$$

In RUN, the first two situations are handled with a call to the *run*() methods of $s1$ and $s2$. This is achieved with the sequence operator of ASM, thereby introducing an intermediate step. The third situation is implied as with SOS.

```
Sequence::run() { s1.run() seq s2.run() }
```

In STEP, the situation is more tricky. The first situation is handled in the *step*() method, where the *pc* is set to the first statement. The second situation is handled in the first branch of the *nextPC*() method, where the continuation after $s1$ is $s2$. The third situation is handled in the second branch of *nextPC*(). The value of $s2$ is retrieved, and set as value of the sequence itself. Then the next step is given by the continuation of the parent.

```
Sequence::step() { RTroot.setPC(s1) }
Sequence::nextPC(node) {
    if (node==s1) then return s2
    else
        let value = RTroot.getValue(s2) in
            RTroot.setValue(self, value)
            return nextPC(parent)
}
```

Finally, we look at the handling of a *method call*, for simplicity with just one parameter. The following situations have to be handled for the semantics. (1) The value of the parameter is evaluated. (2) The value of the parameter is attached to the name of the parameter. (3) The body of the method is executed. (4) The return value of the body is the value of the call.

SOS uses a precondition to handle the first situation, introducing intermediate steps. The second situation is done by extending the name mapping with the appropriate name and value for the call of the body. The third situation is given by replacing the active code. Finally, the fourth situation is implied by the handling of return values in SOS-VM.

$$\frac{< e, v, s > \rightarrow * < \bot, v', s' >}{< \textbf{call}\ m(e), v, s > \rightarrow < m.body, v', s' \uplus \{m.parName \mapsto v'\} >}$$

In RUN, the first situation is covered by a call to the *run*() of the parameter, introducing intermediate steps with the underlying sequence operator (**seq**). The value is attached to the name using the underlying parameter handling of the *run*() method of *body*, which also handles the third situation. Finally, the result value handling is implied by the return result handling of MOF-VM.

```
Call::run() { value := param.run() seq method.body.run(value) }
```

In STEP, the first situation is given by the *step*() method. The second and third situations are handled in the first alternative of *nextPC*(). Here, a new stack frame is created in order to keep the parameter value. Then the program counter is set to the *body*. Finally, the fourth situation is handled in the second alternative of *nextPC*(). The stack frame is removed and the value is attached to the call. The next step is given by the continuation of the parent.

```
Call::step() { RTroot.setPC(param) }
Call::nextPC(node) {
    if (node==param) then
        let sf = new(StackFrame) in
        let value = RTroot.getValue(param) in
            sf.env.add(method.parameter.definition, value)
            RTroot.push(sf)
            return method.body
    else //return from call
        let value = RTroot.getValue(method.body) in
            RTroot.pop() seq RTroot.setValue(self,value)
            return nextPC(parent)
```

The formulation of *run*() and *step*() can also be done using other formalisms with sufficient formality, e.g. UML activities or Java code.

7 Summary

In this paper, we have discussed what operational semantics entails and how it can be defined in an object-oriented setting. We have distinguished between structure semantics and dynamic semantics, where structure semantics describes the instantiation of the language constructs and dynamic semantics involves collections of trivial state changes. For both kinds of semantics, it is essential to rely on an underlying abstract machine providing some kind of semantics. We have used the combination of MOF-VM and ASM as such a machine.

The proposed operational semantics is object-oriented because it gives a clear indication of the runtime structure used with all the objects and their connection to each other. In contrast to standard SOS, it allows to have a clear distinction between the RTE of the language and the RTE of MOF-VM.

The paper has detailed the relation between operational semantics with its two parts and the underlying machine. The semantics is described using methods attached to runtime objects and to syntax objects. It is the task of the language designer to decide where to place the semantic methods.

The RTE provides explicit elements for runtime configurations, thereby solving all the problems as indicated in the introduction. The next step is to turn this approach into a meta-language for defining operational semantics and implementing it in an appropriate tool.

Acknowledgements. We thank the anonymous reviewers for their helpful questions and remarks.

References

1. Börger, E., Stärk, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Secaucus (2003)
2. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J.F.: Rewriting logic and its applications maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2), 187–243 (2002)
3. OMG Editor. OMG Meta Object Facility (MOF) Core Specification Version 2.4.2. Technical report, Object Management Group (2014)
4. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex, 1st edn. The MIT Press, Cambridge (2009)
5. Gjøsæter, T., Prinz, A., Nytun, J.P.: MOF-VM: instantiation revisited. In: Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, pp. 137–144 (2016)
6. Henriksen, J.O.: SLX: the X is for extensibility [simulation software]. In: Proceedings of Simulation Conference, Winter, vol. 1, pp. 183–190 (2000)
7. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987). doi:10.1007/BFb0039592
8. Kleppe, A., Warmer, J.: MDA Explained. Addison-Wesley, Boston (2003)
9. Klint, P., Storm, T., Vinju, J.: EASY meta-programming with rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18023-1_6
10. Mosses, P.D.: Structural operational semantics modular structural operational semantics. J. Logic Algebr. Program. **60**, 195–228 (2004)
11. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011
12. Plotkin, G.D.: A structural approach to operational semantics. Technical report DAIMI FN-19, AARHUS UNIVERSITY (DK) (1981)
13. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic Algebr. Program. **79**(6), 397–434 (2010)
14. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72901-3_12
15. Wider, A.: Model transformation languages for domain-specific workbenches. Ph.D. thesis, Humboldt-Universität zu Berlin (2015)