Human Comprehensible and Machine Processable Specifications of Operational Semantics

Markus Scheidgen, Joachim Fischer

Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{scheidge,fischer}@informatik.hu-berlin.de

Abstract. This paper presents a method to describe the operational semantics of languages based on their meta-model. We combine the established high-level modelling languages MOF, OCL, and UML activities to create language models that cover abstract syntax, runtime configurations, and the behaviour of runtime elements. The method allows graphical and executable language models. These models are easy to read by humans and are formal enough to be processed in a generic model interpreter. We use Petri-nets as a running example to explain the method. The paper further proposes design patterns for common language concepts. The presented method was applied to the existing modelling language SDL to examine its applicability.

1 Introduction

Language specifications, especially the definition of language semantics, are usually either informal or mathematical. These specifications are human readable, even though they might be imprecise or require substantial mathematical knowledge. However, it is normally hard to automatically derive computer tools from such specifications. We want to create model based language definitions that are both: comprehensible to humans and, at the same time, machine executable. Such definitions can be valuable for prototyping new languages, or creating tools for existing languages in a model driven fashion.

Meta-modelling is an already established technology to model the abstract syntax of languages in a human appealing and yet machine processable way. Other modelling techniques (based on meta-modelling) do the same for the language aspects graphical and textual notation, code-generation, or model transformations. Our contribution to the general goal of modelling languages is a method that uses existing graphical (meta-)modelling languages on a high level of abstraction to define operational semantics. We formally describe languages and can therefore execute models solely based on the according language definition by using a generic model interpreter.

Plotkin's structural operational semantics [1] is the standard way to define the operational semantics of programming languages. It uses transition systems

$\langle \Gamma, \rightarrow \rangle$, where Γ is a set of configurations γ and $\rightarrow \subseteq \Gamma \times \Gamma$ are the possible transitions between configurations. To define the operational semantics of a meta-model based language we use its meta-model M to define sets of models Γ_M. These models act as configurations. Furthermore, we use actions over models as transitions from one model of M to another model of the same M ($\gamma \in \Gamma_M \rightarrow \gamma' \in \Gamma_M$). We use UML activities in combination with OCL to describe the actions to be executed. These activities describe sequences of model configurations: $\gamma \rightarrow^* \gamma'$. Meta-models and activities form language models which define abstract syntax and operational semantics. We developed a generic model interpreter that can process such language models. This tool interprets an input model γ_{in} by changing it as defined by the activities in the corresponding language model. The result is a model that evolves according to the specified operational semantics: $\gamma_{in} \rightarrow^* \gamma$.

In the next section we continue with related work. Section 3 explains the basic concepts of our method and shows an example language model which describes the operational semantics of Petri-nets. In section 4 we show that more complex languages need to distinguish between elements that describe abstract syntax (define the models that the user can write) and runtime elements (describe additional information that is necessary when a model is executed). Section 5 discusses reusable designs for operational semantics and presents a pattern for instantiation as an example. Section 6 briefly describes the application of our method to the modelling language SDL and thereby reasons that the framework is applicable and scales up to practical languages. The paper closes with conclusions in section 7.

2 Related Work

Work on generated or generic language tools includes frameworks for the development of domain specific languages. These frameworks use meta-models as the core of language specifications and also cover language aspects like notation, analysis, transformations, or operational semantics. Such frameworks are GME [2], XMF [3] (originated in the MMF approach [4]), AToM3 [5] and meta-programming facilities like MPS [6], kermeta [7], AMMA [8] MetaEdit+ [9]. Some of these frameworks define semantics through general purpose programming languages (MPS, MetaEdit+), others provide specialised languages to define semantics (XMF, kermeta, AToM3). Two different approaches to semantics can be identified: GME and AToM3 use model transformations into a different language or formalism (semantic domain). AMMA, Kermeta, XMF, and MPS use an action language to define operational semantics.

There are several approaches using a specific meta-language for the definition of operational semantics. In [10] Engels et al present a graphical modelling approach for UML semantics based on collaboration diagrams and graph transformations. This approach provides strong mathematical foundations, but results in very verbose semantic rules, which are hard to read and execute. In [8] Abstract State Machines (ASM) are integrated into the DSL framework AMMA to

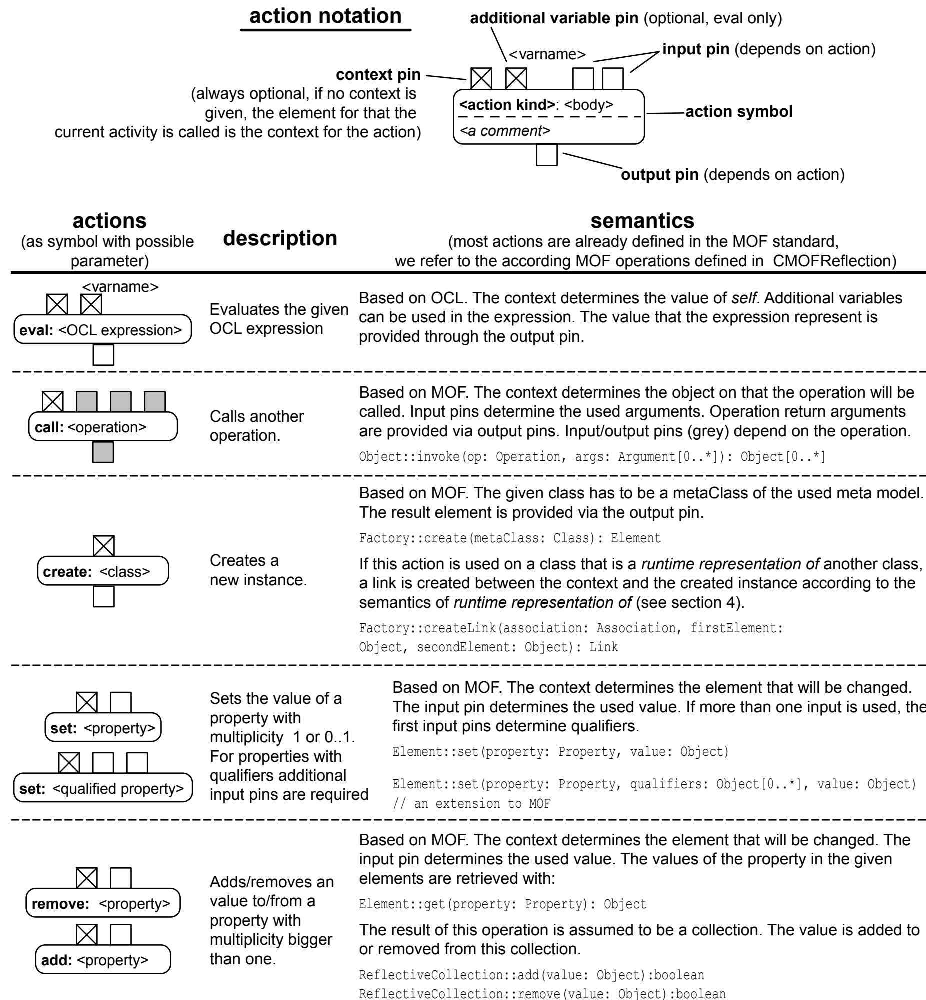

Fig. 1. A list of actions that can be used to define operational semantics.

support specification of execution semantics for DSLs, using ASMs as just another DSL. Muller et al [11] use a textual action language in combination with OCL for high level semantics descriptions. This action language is executable and provides the foundation for the DSL framework kermeta [7]. A similar approach is used in Mosaic [3] which uses an OCL version extended with actions to define language semantics. We recycled the idea of using OCL for expressive model navigation in our approach. In [12] Gerson Sunyé et al explore the possibility of UML action semantics [13] to create executable UML models and already suggest the use of activities with action semantics for meta-modelling.

We use this idea and reduce the set of actions to those necessary to describe operational semantics based on model changes.

We use a CMOF based modelling architecture as foundation for our approach. The reason is that the *CMOF* meta-meta-model [14] provides means for feature refinement in the context of class specialisation, which allows better expressions of abstractions in meta-models than EMOF or similar models (MOF 1.x, EMF-Ecore). The according MOF features were formalised by Alanen and Porres in[15], and we provided a programming framework for *CMOF* based modelling in [16].

Along with structural operational semantics [1], semantics are traditionally defined based on grammars for abstract syntax. A formalism, like term re-writing, is used to describe manipulation of abstract syntax trees (AST are instances of grammars). This describes interpretation of an input program rep-resented by an AST. The formal SDL semantics definition [17] uses Abstract State Machines (ASMs) to realise a similar approach: it defines abstract syntax and runtime states with grammars and represents corresponding ASTs as evolv-ing algebras manipulated by ASMs. Our approach replaces grammars/signatures with meta-models, ASTs/algebras with models, and re-writing/ASMs with our combination of activities, OCL, and actions.

3 Basic Concepts

Operational semantics describes transitions between models (configurations). Such transitions can be realised by changing a model (evolving configuration). To describe and execute operational semantics defined with such transitions we need: (1) changeable models; (2) types of transitions, in our case atomic model changes, which we call actions; and (3) a language to control what action is to be executed under what conditions and in what order.

Models, as instances of meta-models, aren't normally supposed to change. An UML model, for instance, does not change once it is written. But because models constantly change during editing, MOF already supports model changes. We can dynamically create new elements or update attributes of existing elements to change a model. We extended MOF's CMOF model with property *qualifiers* (as defined in UML). We use this extended CMOF language for our meta-models.

Fig. 1 defines a fixed set of atomic actions which we use in UML activities. The semantics for these actions is given by the MOF standard. The semantics for UML activities (as we use them) is founded on Petri-nets as described in [18]. Activities are connected to the meta-model via operations. The behaviour of each operation in a meta-model can be implemented with an activity. When a operation is called, the according activity is interpreted. Meta-models are object-oriented models, and calling a operation means that it is called on an instance of the corresponding class. This also means that activities are always interpreted in the context of an object. This context can be addressed with the value *self*. Operations can also have parameters, and calling an operation requires according arguments, which can be used in the activity.

Each model can be executed like a normal object-oriented program by calling an operation and interpreting the according activity. One operation has to serve as a dedicated *main* operation. There is usually a model element, known as the outermost composite, which contains all other elements. It is reasonable practice to define the *main* operation in the class that describes this element.

3.1 An Example Language – Petri-Nets

In this section we demonstrate our meta-modelling method and create a language model for Petri-nets. This model consists of descriptions for an abstract syntax and an operational semantics for Petri-nets. Fig. 2 shows this language model and an example Petri-net. We choose Petri-nets as an example language, because they have a very small abstract syntax and simple but clear semantics.

The language model specifies that a Petri-net consists of places and transitions. Places can be related to transitions, and each transition has an arbitrary number of input and output places. A place can contain any number of

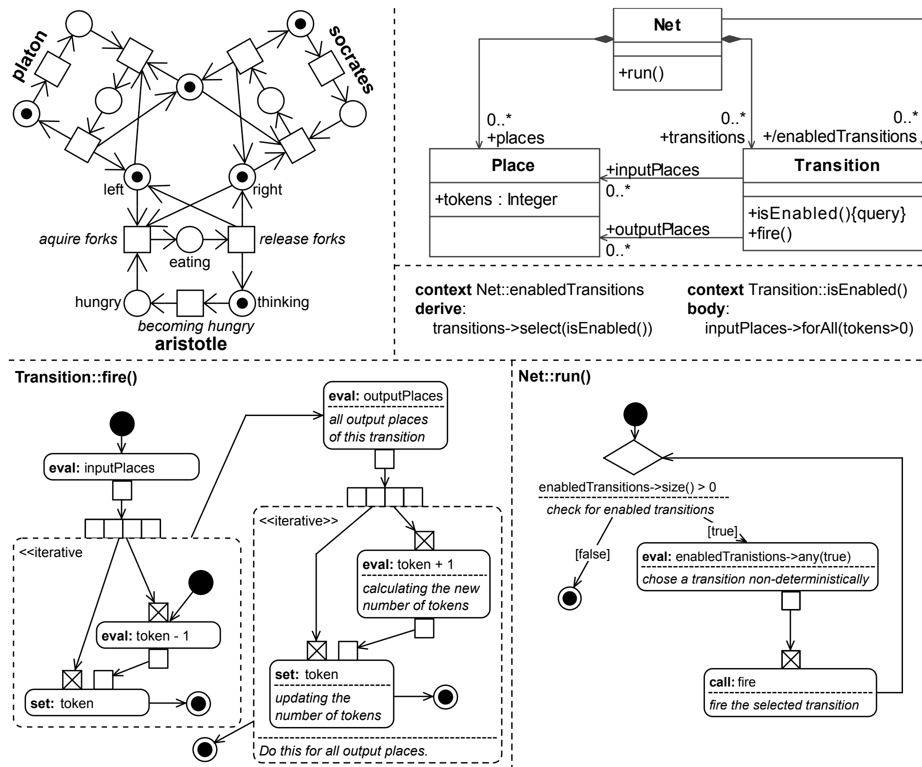

Fig. 2. Petri-nets as an example: an example net and a language model for Petri-nets containing an abstract syntax model, OCL expressions, and activities.

tokens. The figure also shows an example Petri-net, an instance of the given meta-model. This Petri-net diagram of the famous dining philosophers uses the typical Petri-net notation: places are drawn as circles, transitions as boxes. Incoming arcs show the input places of a transition, and outgoing arcs show their output places. Dots inside places show the number of tokens in a place. All text in this Petri-net diagram is commentary, and no text fields are defined in the meta-model. However, we will use these names in further explanations. Please note that we mixed Petri-net structure (places and transitions) with Petri-net configurations (tokens). We will address this issue in section 4.

The semantics of Petri-nets is simple. Transitions are the only active elements in a net. They change the number of tokens in places, which are the only dynamic elements in a net. A transition changes the number of tokens in its input and output places when it is fired. But a transition may only be fired when it is enabled, and it is enabled when all its input places contain at least one token. Given these definitions, a Petri-net has the following semantics: a transition is chosen from all the enabled transitions non-deterministically. The chosen transition is fired. This means that the number of tokens in all input places is reduced by one, and the number of tokens in all its output places is increased by one. Transitions are chosen and fired until the net contains no more enabled transitions.

We describe the operational semantics with operations and derived properties. The Petri-net meta-model contains two operations, one query operation, and the derived association end `enabledTransitions`. These elements realise the informally explained semantics in a formal and executable way. The query operation and derived property can be fully determined by OCL expressions. These elements need no further refinement or implementation; the OCL expressions can be evaluated by the computer right away. `Transition::isEnabled()` returns *true* when the transition contains tokens in all input places. The derived association end `enabledTransitions` selects the collection of all enabled transitions in a net. The OCL expressions are given in fig. 2.

The behaviour of the other two operations can be specified using the activity language (see fig. 2). Imagine that the operation `Transition::fire()` is called for the transition *becoming hungry* during the execution of the example net. The first action is to evaluate the expression `inputPlaces` in the current context *becoming hungry*. This transition has only one input place: *thinking*; the result is a collection containing *thinking* only. After that, the collection is iterated. The activity in the iterative expansion region is executed for each element; in this case this is only *thinking*. This sub-activity evaluates `token`-1. This time *thinking* and not *becoming hungry* is used as context. The value $(1 - 1 = 0)$ is the result and is set to the property `token` in the context of *thinking*. After that is done, the number of tokens in each output place is increased in a similar fashion.

The operation `Net::run()` acts as *main* operation; it executes the net. This means it fires enabled transitions as long as there is at least one enabled transition left. A decision is used to continue or stop based on whether the set of

`enabledTranitions` is empty or not. When it is not empty, one transition is selected non-deterministically, using OCL's `any`. After that, `Transition::fire()` is called on the selected transition.

4 Distinguishing between Syntax and Runtime Elements

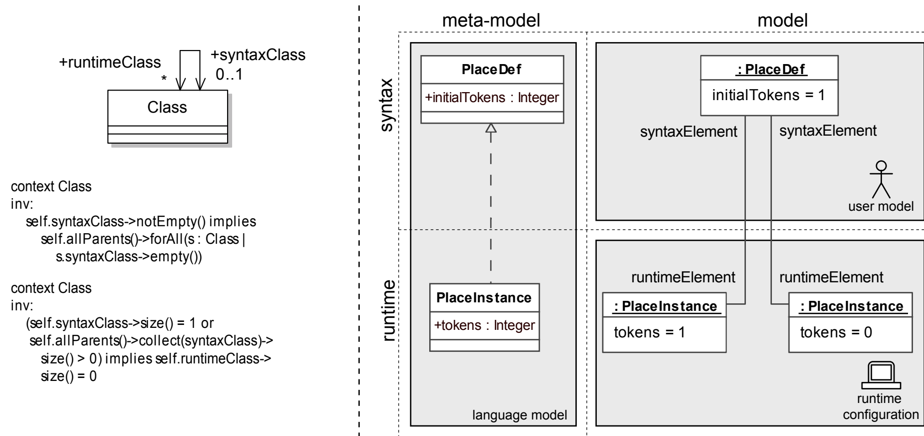

Fig. 3. A new meta-model relationship to relate syntax and runtime elements with each other.

In the last section we defined operational semantics by describing model changes. All runtime information needed to execute a model could be stored within the model. This approach has two flaws. One, in general we need additional data structures to describe a runtime configuration. A program, for example, is only one part of a configuration during a program run. Other parts are slots for variable values, heap memory, and program counters. The second problem is that when we change the input model it will be lost for future execution. In the moment we destroy a token in one Petri-net place and create it in another, we destroy the original marking. When we say the initial marking is part of the Petri-net, we would destroy the net by executing it. We should have stored the actual number of tokens independent from the initial number of tokens.

To describe complete configurations, a meta-model has to define both: the abstract syntax of the language and additional data-structures needed for runtime information. We distinguish between syntax classes and runtime classes. The set of all syntax classes describes what users of the language can write in their models. The set of all runtime classes describes data that can be created and used during the execution of a model. Syntax and runtime elements can be related to each other.

Fig. 3 shows (on the left side) an extension of the MOF meta-meta-model as a meta-model for a new relationship between classes. We call this relation *runtime representation of*. This directed relationship indicates that one class denotes a runtime representation of a syntax class. We use the UML realisation arrow (which has no predefined meaning in MOF) to notate this relationship. Fig. 3 also shows two corresponding OCL constraints that limit the use of this relationship: there are no circles allowed and a class cannot be runtime representation for itself.

The right side of Fig. 3 shows an example of a runtime representation: a `RuntimePlace` is the runtime representation of a `PlaceDef`. The language user, who creates the Petri-net, determines the initial number of tokens using the corresponding slot in `PlaceDef` instances. At runtime, the numbers of tokens are stored separately in `RuntimePlace` instances. That allows us to run the same net in two runtime representations at the same time. The *runtime representation of* relationship will allow to navigate between instances of runtime and corresponding syntax classes. The `create` action described in the previous section, will automatically link a newly created instance of a runtime class with the corresponding syntax class instance.

4.1 An Advanced Example Language – Hierarchical Petri-nets

In this section we use a more sophisticated Petri-net variant to demonstrate that most semantics descriptions require to differentiate between syntax and runtime elements. In the previous section, we modelled a dinner table with three philosophers. This model already contained the same philosopher pattern three times. We model *Hierarchical Petri-nets* (also known as *modular Petri-nets*, not to be confused with Petri-nets that use sub-nets as tokens), which allow to build an abstraction for this pattern. We can model the common philosopher behaviour once, and use it for multiple philosophers. Fig. 4 shows such a hierarchical Petri-net for the dining philosophers.

Hierarchical Petri-nets contain additional concepts and notations. We can define sub-nets, notated as a smaller net inside a box. These sub-nets have dedicated interface places. In the example the behaviour of a philosopher is modelled as a sub-net. The places for his left and right fork are interface places, because each philosopher has to share this place with his right and left neighbour. In hierarchical Petri-nets each net can contain sub-net usages which are notated as a black box. Petri-net usages are related to regular places to connect interface places with real places. These connections are drawn with lines that have the respective interface place name written on them.

Since hierarchical Petri-nets contain additional concepts, we also need a different language model (fig. 5) with additional classes and different descriptions of operational semantics. We have to distinguish between the definition of a sub-net and the usage of a sub-net. `NetDef` represents Petri-net models. `NetDef` instances can contain transitions, sub-net definitions, sub-net usages, and places. Net usages are realised in the class `NetUsage`. Instances of `NetUsage` reference a `NetDef` to characterise the used sub-net. The former class `Place` is now called

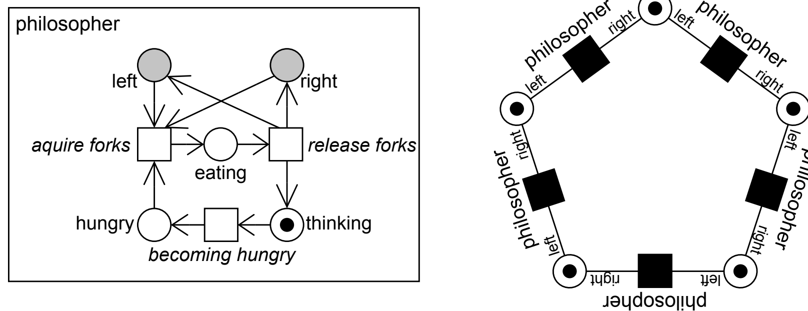

Fig. 4. A hierarchical Petri-net for the dining philosophers.

PlaceDef. Instances of this class are used to model places; we will need another place class to represent places at runtime. The connection of interface places is modelled as a qualified property of NetUsage. A qualified property works like a map. In this case, it associates a NetUsage with a PlaceDef based on another PlaceDef as key: a usage is connected to places, and each of those connections is qualified by an interface place.

Hierarchical Petri-nets use one sub-net several times. We use several instances of the same net definition to to store the number of tokens in each sub-net instance separately. We cannot use the semantics definition from the place/transition Petri-net example, because the places in one sub-net are now used several times in multiple usages of the same sub-net. When the number of tokens in a place of one instance changes, it would also change in the same place of all the other instances.

The definition classes NetDef and PlaceDef are syntax classes. They are used within the Petri-net model; they are classes for things the user draws in a Petri-net diagram. The dining philosopher model is a NetDef instance, the philosopher sub-net is a NetDef instance; all places in the model are PlaceDef instances. The other two classes RuntimeNet and RuntimePlace are runtime classes. A RuntimeNet can contain instances of sub-nets (other RuntimeNet instances) and contains RuntimePlaces using a qualified property with the according PlaceDefs as keys.

When a user provides a hierarchical Petri-net it will only contain instances of the syntax classes. Creating runtime class instances is part of the semantics. It is part of the semantics to initially instantiate the dining philosophers Petri-net, create sub-net instances for all the usages of philosopher. This instantiation task is modelled in the operation NetDef::instantiate. This operation will create a runtime representation of itself and all its contained places; it will furthermore create runtime representations of all used sub-nets recursively and connect its interface places to real places. Fig. 5 shows the activity diagram for this operation.

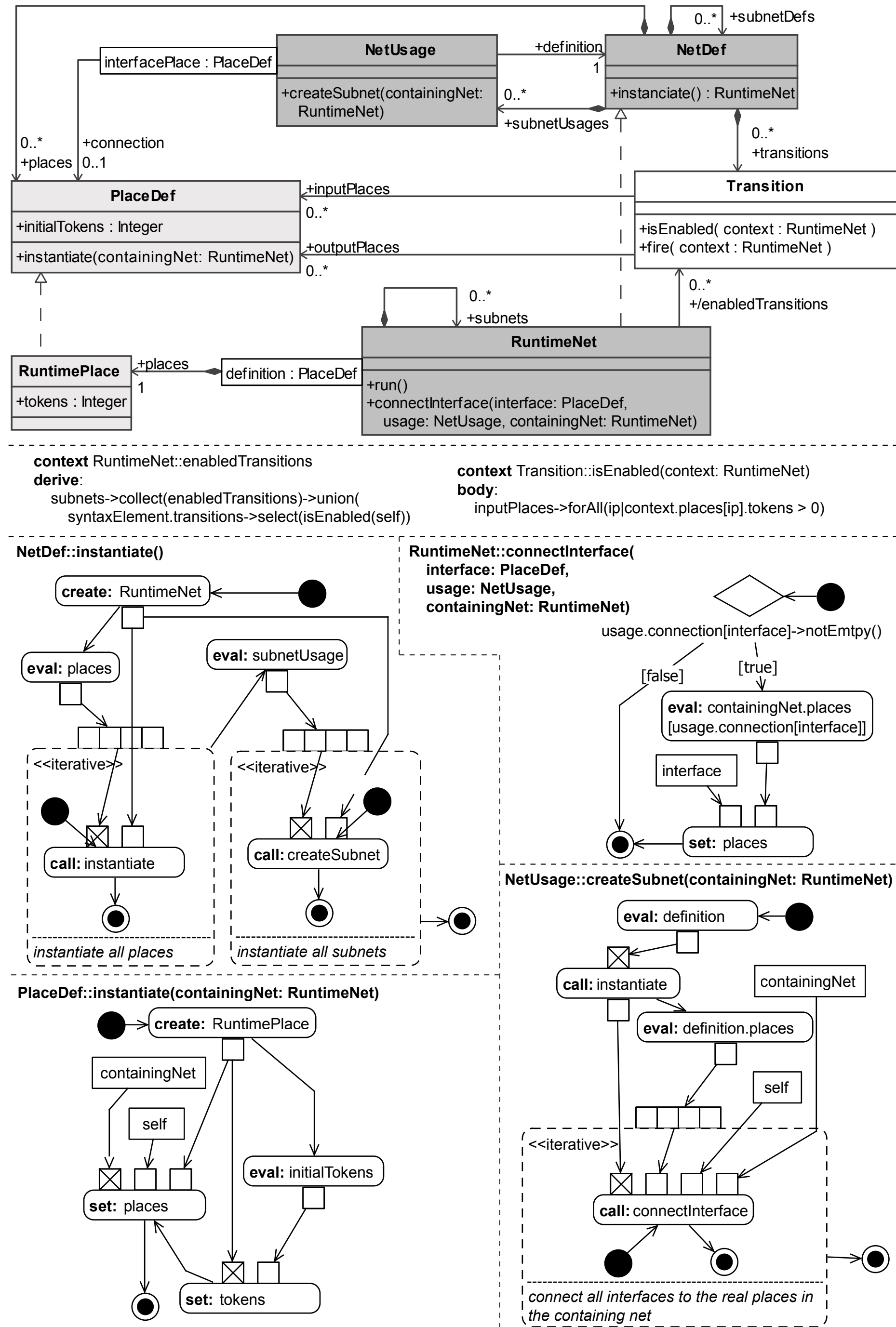

Fig. 5. A Language Model for Hierarchical Petri-nets.

After `NetDef::instantiate` was called for the top-level Petri-net, we can use the created `RuntimeNet` by calling its `run` operation. Even though `run`'s signature hasn't changed from the previous section, it works a little different due to the changes in the meta-model. Transitions can only be fired in the context of a `RuntimeNet`. Since transition is only a syntax class with no runtime counterpart, it is also only related to `PlaceDef` (the syntax class for places). The input and output places of a transition are instances of `PlaceDef` and the number of tokens cannot be accessed or changed directly on them. The operations of transition have to access the corresponding `RuntimePlace` using a `RuntimeNet` as context. The run operation itself (not shown) also works different: it still choses one transition from all enabled transitions. But because one transition can be enabled in different sub-nets (in the starting configuration, *becoming hungry* is enabled in all five philosophers) `run` must also chose one of these sub-nets that the chosen transition is enabled in. After transition and `RuntimeNet` are chosen, run fires the transition using the chosen `RuntimeNet` as context argument.

5 Language Design Patterns

Patterns in software engineering form a basis for reusing working designs [19,20]. We want to use patterns for language modelling. A language design pattern describes an abstract language concept. We implement these patterns as abstract libraries (similar to the abstraction libraries in the UML meta-model). Each of these pattern implementations consists of abstract classes for syntax and runtime elements as well as their operations' behaviour. The pattern implementations can be used by usual object-oriented means: a general pattern class is specialised and its features refined to fulfil a specific purpose in a concrete context.

Fig. 6 shows such a pattern implementation (white classes) and how it is used by specialisation in two examples (grey and dark grey). This pattern describes the abstract concept instantiation. It defines `Classifiers`, which define sets of instances with common attributes defined as `Features`. These features can have a type. A `Classifer's Instance` provides a `Slot` for each `Feature`. Each `Slot` can hold `Values` of the corresponding `Type`.

This pattern is common to many languages, including MOF and UML. Without knowing it, we already used this pattern in the previous section, where we had `NetDef` (classifiers of sub-nets that can be instantiated at different places) and its runtime counterpart `RuntimeNet` (runtime representations of sub-nets). `NefDefs` have `PlaceDefs` as features and `RuntimeNets` have `RuntimePlaces` as corresponding slots. We don't need the type/value part of this pattern, because the number of tokens is always stored as an integer. But here we could extend the language if we wanted to introduce objects for tokens as in *Object Petri-nets*.

The other example application for this patterns are `Classes` and `Fields` (also known as member variables) in object-oriented languages such as Java. This application of the pattern also uses `Types` and `Values`, even though we simplified the problem in this example: classes are the only types and conclusively objects the only values. Another application for this pattern are procedure-like concepts.

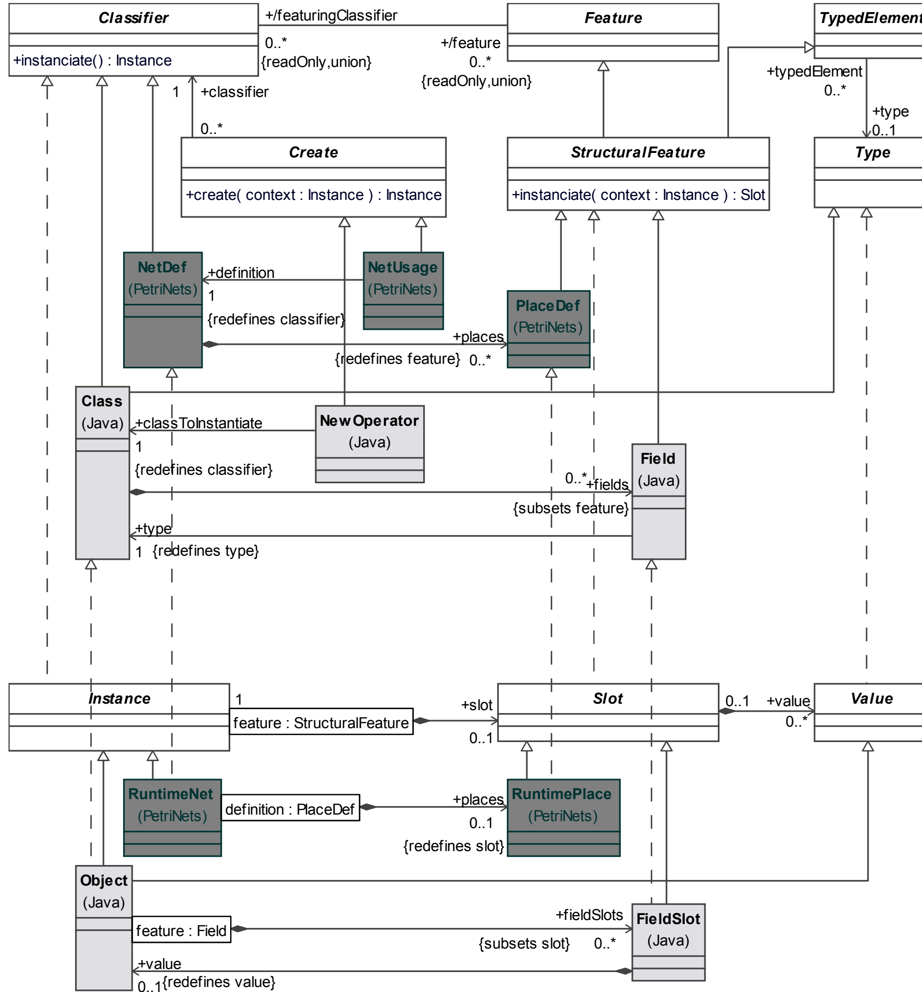

Fig. 6. A general pattern for classifier and instances.

Procedures are classifiers, parameters and variables are features, call frames are procedure instances with proper slots for variable or parameter values.

6 SDL: A Case Study

In order to reason about the applicability our meta-modelling method, we applied it to SDL—the Specification and Description Language[21]. This is an existing graphical modelling language, widely used in the telecommunication sector. It is similar to UML but has unambiguous semantics. SDL supports structural modelling, similar to UML components, provides a data-type definition language,

and allows behavioural modelling with concurrent processes, signal-based communication, and state charts.

We described the use of tools in language specifications in [22] and presented a general architecture for the meta-model-based specification of SDL and related languages in [23]. We created an experimental tool-chain for SDL, including a parser, semantics analyser, model transformations, and code-generator. We use the method from this paper to create a simulator for SDL specifications [24].

To define the operational semantics, we created a meta-model which includes runtime classes, and according operation implementations for SDL. We started to define the semantics for a representative subset of SDL. This meta-model already contains 108 classes with 257 properties and 105 operations. We composed the SDL meta-model from several design patterns. These patterns were realised in abstract libraries, which were (re-)used several times throughout the SDL language model. 30 classes are part of pattern implementations and 78 are SDL specific classes. In a first prototype we specified the operation's behaviour with Java, which is now replaced more comprehensible activities.

Three pattern implementations are used in the SDL model: the *instantiation* pattern, introduced in the previous section, a pattern for *concurrent processes and communication*, and a pattern for the *evaluation of terms*. The structural part of SDL is dominated by the instantiation pattern, because SDL-structures are defined by object-oriented classifiers, called *agents*. Agents can be recursively composed: an agent instance (instance) is a feature of another agent type (classifier). Agents can be connected through communication channels and gates. This is a combination of two patterns. The instantiation pattern is used to describe agent type and instance relations and the concurrency pattern describes interaction of different agent instances. The SDL behaviour is characterised by state machines and statements. State machine behaviour, the triggering of transitions, is realised as part of the concurrency pattern: communication as synchronisation of processes (processes in SDL are nested state machines as part of agent instances). Statements, the other part of SDL behaviour, are similar to other imperative programming languages. Expressions and data types used in those statements are realised with the evaluation pattern.

The SDL language model can be executed with our generic model interpreter. Input SDL specifications are transformed into a model representation according to the SDL language model using the tools presented in [23]. The generic interpreter runs this input SDL specification: it initially creates a runtime configuration for the specification and changes it during execution. As part of the defined operational semantics, *Message Sequence Chart* models are created from the changing runtime configuration to visualise the running SDL system.

7 Conclusions

We combine MOF meta-models with an action language based on UML activities and OCL to create language models that contain definitions for abstract syntax and operational semantics. We developed a generic model interpreter which can

be configured with language models. It takes a model as input and executes the model based on the semantics defined in the language model. The used languages allow human readable graphical models of language structure and operational semantics. These language models are at the same time formal enough to be machine interpretable. With such characteristics, the method is ideal for language prototyping, creation of reference tools, and the development of domain specific languages. We created a language model for a subset of SDL. This experience showed that our method scales up to a practical language of reasonable size.

We have all tools necessary to create and interpret language models. We use a normal UML case tool (class diagrams in MagicDraw) to define the structure part of language models. We augment MagicDraw models with activities using a graphical editor, specifically developed with GEF (eclipse). As a future work, we are developing a runtime environment based on the abstract eclipse debugging plug-ins. This should allow to support the development process of language models (debugging meta-models) and also provide generic debugging facilities (debugging models).

We have to critically admit: the assessment that operational semantics modelled with our approach results in more human readable language specifications than comparable techniques (ASMs, mathematical semantics definitions, or even natural language text) is purely based on the graphical modelling argument and our prejudice experiences with it. This hyphothesis has yet to be proven by either more representable experiences or sound usability evaluations. But formal graphical models of operational semantics could play the same role that normal meta-models play for the definition of abstract syntax.

On the machine execution side, we have disadvantages and advantages. Unfortunately, but as expected, model execution, based on language models and our generic tool, compared to equivalent hand crafted tools performs less by magnitudes. It is future work to analyse and antagonize the reasons for that. An advantage, however, is that we have a meta-model based representation of the model's runtime state. Besides the fact that we can execute models right away, we can use other modelling techniques to analyse these runtime states, e.g. define constraints over them, or use model transformations on them to create different representations, e.g. record test cases.

References

1. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
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 (2003)
3. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodeling, A Foundation for Language Driven Development. Xactium (2004) http://www.xactium.com.
4. 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. (April 2001)

5. The Modelling, Simulation and Design lab (MSDL), School of Computer Science of McGill University Montreal, Quebec, Canada: AToM3 A Tool for Multi-Formalism Meta-Modelling. http://atom3.cs.mcgill.ca/index.html.
6. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard, electronic monthly magazin (November 2004)
7. Team, T.: Triskell Meta-Modelling Kernel. IRISA, INRIA. www.kermeta.org.
8. Ruscio, D.D., Jounault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of SDLs (2006) technical report.
9. Case, M.: MetaEdit+. http://www.metacase.com.
10. 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: UML 2000 - The Unified Modeling Language. Advancing the Standard.
11. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Model Driven Engineering Languages and Systems: 8th International Conference. LNCS, Springer (2005)
12. Sunyé, G., Pennaneac'h, F., Ho, W.M., Guennec, A.L., Jézéquel, J.M.: Using UML Action Semantics for Executable Modeling and Beyond. In: 13th International Conference on Advanced Information Systems Engineering. LNCS, Springer (2001)
13. OMG: Action Semantics for the UML. Object Management Group (2001) ad/2001-08-04.
14. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group (October 2003) ptc/03-10-04.
15. Alanen, M., Porres, I.: Basic Operations over Models Containing Subset and Union Properties. In: 9th International Conference Model Driven Engineering Languages and Systems. LNCS, Springer (2006)
16. Scheidgen, M.: CMOF-Model Semantics and Language Mapping for MOF 2.0 Implementations. In: MBD/MOMPES, IEEE Computer Society (2006)
17. ITU-T: SDL formal definition: Dynamic semantics. In: Specification and Description Language (SDL). International Telecommunication Union (November 2000) Z.100 Annex F3.
18. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Software Engineering. (2005)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse in Object-Oriented Designs. In: Proceedings of ECOOP'93, Springer-Verlag
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. 1st edn. Addison-Wesley Professional (1995)
21. ITU-T: ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunication Union (August 2002)
22. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based Language Development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (November 2004)
23. Fischer, J., Kunert, A., Piefel, M., Scheidgen, M.: ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In: SDL 2005: Model Driven: 12th International SDL Forum. LNCS, Springer (2005)
24. Systeman Alysis and Modelling Group, Department of Computer Science, Humboldt-Universität zu Berlin: An Operational Semantics Model for SDL www.informatik.hu-berlin.de/sam/meta-tools/sdl.