A Set-Theoretic Formal Specification of the Semantics of Kermeta

Moussa Amrani

Laboratory for Advanced Software Systems (LASSY)
University of Luxembourg
6, rue R. Coudenhove-Kalergi
L-1359 Luxembourg

TR-LASSY-12-12

October 2011 — May 2012

Contents

Chapter 1

Introduction

Model-Driven Engineering (MDE) is a novel approach to Software Development. This approach elects models as first-class artifacts during all phases of development while having as main concern to improve and maximise the productivity in short and long term. The goal is to improve and maximize the productivity in both short and long term. In a short term, by increasing the average functionalities a software system is delivered with; in a long term, by reducing the costs of maintenance and accuracy of the systems by reducing their sensitivity to change (coming either from technologies, or from stakeholders) [3].

From a methodological point of view, MDE tries to use models throughout the development of complex systems to improve the effectiveness of every days tasks, such as documentation of the system at early stages, specification of the different components of the system, design of the overall architecture, development itself, versionning, maintenance, etc. From an engineering point of view, models are used to provide a team involved in the development process the most accurate level of abstraction to deal with the essential complexity of each task and let all the machineries behind the tool support deal with (most of) the accidental complexity, by using transformations to manipulate models [6].

MDE can be seen as the natural continuation of a movement that has its root in the Computer Science itself: raising the abstraction level to allow programmers interact with machines using their own thinking mechanisms instead of communicating at the level of the machine. Domain-Specific Modeling Languages (DSMLs) use the MDE methodology, but focus on a particular expertise domain. They take the opposite philosophy of what is intended with General-Purpose Languages: by focusing on a particular domain, they discard by nature the possibility of reuse in areas that go beyond the domain of interest. However, thanks to this restriction, DSMLs allow domain experts to achieve a better degree of automation when getting the final result without knowing low level details.

From a practical point of view, DSMLs try to embed the domain's concepts and their links, as well as the rules that governing them. This way, they offer the possibility for programmers or experts to deal with the same notions they are used to in their daily life. Most of the time, DSMLs are manipulated through visual syntaxes that try to reflect thoroughly the visual representation of these concepts. Ideally, this should allow experts to be able to create their own models without the help of engineers.

However, from a theoretical point of view, DSMLs are still languages, and it should be possible to study their essence with the classical tools available from research and practice on formal languages. Therefore, a DSML should possess the two core components of every language [17]: several *syntaxes*, among which one is abstract, and others that are concrete, serving as interfaces with the experts; and a *semantics*, capturing the intended meaning of the DSL in terms of a semantic domain.

We propose to study the theoretical implications of the definition of DSMLs, by mathematically characterising what a DSML is, and providing a formal description of its components. Formalising the idea of DSML is helpful in several concerns: it provides a common ground for comparing, evaluating and engineering DSMLs; and it offers a practical and manipulable knowledge about DSMLs that serves the construction of associated tools accompanying DSMLs.

The study focuses on Kermeta [21, 19] for precise reasons:

1. it covers all aspects of DSML definition;
2. it is based on the Object Management Group's (OMG) standard language MOF [22] for the structural aspects and deals with transformation through an Object-Oriented (OO) action language;

3. it is involved in a joint effort to bring MDE technology into the industrial field, making it effectively used for industrial cases [34]; and

4. it has gained maturity, stability, and is backed up by an important user community.

To address the previous objectives, we propose a formal specification of the Kermeta language, with the following contributions:

- The core contribution of this paper is the full formal specification of the semantics of a consequent subset of Kermeta's structural (SL) and action languages (AL), in a precise, concise and homogeneous way: to the best of our knowledge, only partial Kermeta formalisations, or formalisations of some aspects only, are contributed in published literature.

- The proposed formalisation does not relies on any tool-oriented syntax or formalism, avoiding the reader to learn another formalism in order to understand the specification. Instead, we use only standard mathematical techniques in the area of semantics: set-theoretic constructions and rewriting systems to describe a Structural Operational Semantics (SOS, cf. [35]). We believe that this approach is pedagogical and can serve as a reference; whereas practitioners can start from it and exploit at implementation time tools specificities when needed.

- This work can first be seen as a contribution to the effort of the community to provide solid foundations for MDE frameworks. Moreover, this work takes place in a general effort to apply formal verification techniques in the area of MDE, which is especially useful in safety-critical application domains. Specifying formally the semantics of such a language is a necessary, yet non trivial step towards this goal, and since Kermeta uses an OO action language, we believe that adapting existing verification techniques could also be useful.

It is well-known that defining the execution semantics of a particular DSL is not trivial. We propose here to formalise the complete semantics of a metamodeling framework, i.e. a language that describes DSLs and their semantics, which is one level above. This task is usually the first step towards the formal analysis of DSL models and their transformations.

By following this path, we gain a substantial advantage comparing to other approaches: instead of building *ad hoc* formalisations redesigned every time a new DSL is defined, our mathematical framework is defined once and for all, making all DSL defined within Kermeta beneficiate from this formalisation. Of course, this approach has the dual drawbacks of the *ad hoc* approach: First, Kermeta relies on a computation model that is, by nature, asynchrounous, non real-time, non concurrent: any DSML that integrates these aspects will not fit in our formal framework, simply because it is hard to model it in Kermeta. Second, such formal framework rarely stands on its own: it is the first step towards formal analysis that usually requires to discover dynamic properties that hold at execution of DSLs transformations. Because the formalisation has no precise information on what a DSML represents, one has to formulate hidden properties coming from the expertise domain, and build clever abstractions on the transformations to have a chance to tackle formal analysis. Fortunately, in the case of Kermeta and its Object-Oriented AL, there exist plenty of work in this area from which we directly benefit.

This report is organised as follows: Chapter 2 sets background about Kermeta and provides a running example; Section 3 presents the formal specification of the Structural Language, used in Section 4 for the semantics of the Action Language. Section 5 gives some implementation details and lessons learned from it whereas Section 3.7.2 gives perspective by analysing some related works. Section 6 concludes with final remarks.

Chapter 2

Background

In this Section, we provide basic background about Kermeta for better understanding our design choices and the scope of our work. It continues by defining a small running example used to illustrate the mathematical framework defined in this paper. It finishes by defining some ground notations used throughout the paper.

2.1 Kermeta

Kermeta is a well-known metamodeling framework that emerged around 2006. It covers all steps required for building DSLs [12, 21, 19]: a DSL syntax is expressed through a metamodel by using Kermeta's Structural Language; its constraints can be expressed by a Constraint Language aligned with OMG standard OCL; and its behavioral semantics can be captured with the help of Kermeta's Action Language.

We first focus on Kermeta's Structural and Action languages, which are at the center of this contribution. We then provide some general overview on the Kermeta platform to give an idea of how rich the Kermeta framework is.

2.1.1 Kermeta's Languages

One of the first requirements in the design of Kermeta languages was to follow accepted standards. As a matter of fact, Kermeta's Structural Language is a conservative extension of the OMG standard MOF [22] (more exactly, EMOF, but at this paper's level of abstraction, we consider MOF, EMOF and ECORE as synonyms): in particular, *multiple* inheritance for metamodeling can be used. Kermeta enhances EMOF with three interesting capabilities:

Genericity (also known as *parametric classes)*, a notion not yet available in MOF, is a powerful paradigm for achieving concise structural description and reuse.

Model Type is a powerful extension to MOF type notion that allows one to consider a whole model as a type, instead of being restricted to classes enumerations and primitive types. This also achieves high-level abstraction, especially when combined with operation calls using parameter representing models.

Aspects is a convenient way of expressing cross-cutting concerns in a simple and uniform fashion, and then combine them properly. This feature introduces modularity in the metamodelisation process.

These features, except perhaps the notion of model types, are not new: they already exist for example in the object-oriented paradigm. Since their formal grounds are mostly still an open research problem (cf. for example [8]), they are not addressed in this formalisation, but are left for future work (among which aspects require formalisation because they are common in Kermeta behavioural specifications).

The Action Language consists in enriching operations' bodies with statements in an OO fashion: navigation capabilities, class instance creation, basic iterative constructions (like conditionals, loops and assignment), operation calls with a simplified form of polymorphism, generic operations including parameters typed as models, and exception handling. Besides, Kermeta offers the possibility to directly call native Java methods, which is specially useful when dealing with intensive algorithmic computations or interfacing with other tools like databases or internet servers. In our formalisation, all Action Language's construction concerning genericity or model types are obviously discarded, due to the restriction

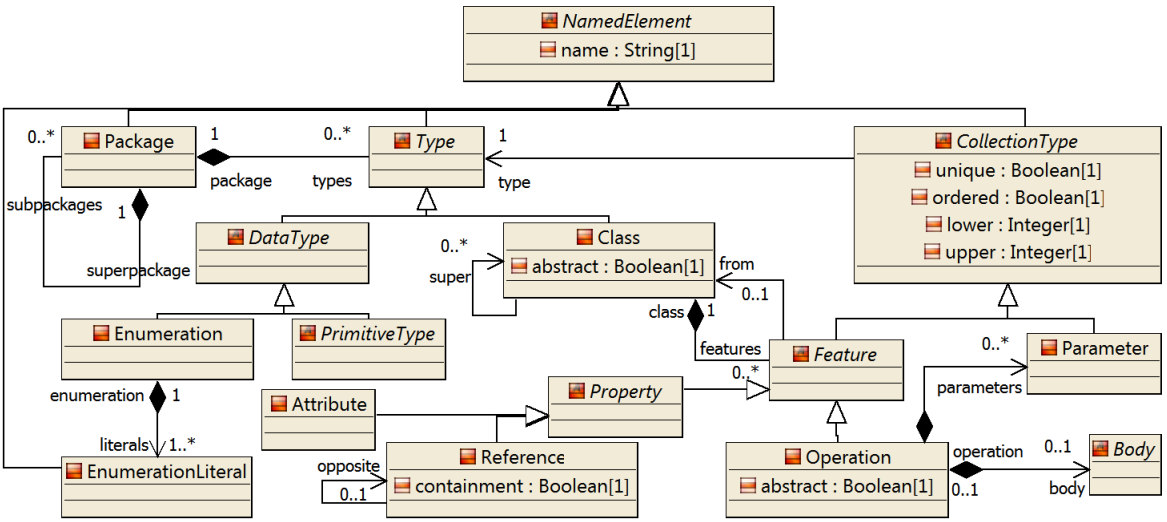


Figure 2.1: Metamodel of Kermeta's Structural language (from [12])

we impose on the Structural Language; moreover, we do not deal with exception handling, since it only concerns transformations that do not behave correctly.

With respect to MOF, Kermeta naturally deals with multiple inheritance, which raises well-known issues in the context of object-orientation: conflicts appear when more than one inheritance path defines a feature sharing the same name. We summarise the discussions that already appear in Kermeta's Manual [12].

From a metamodeling point of view, it does not make sense to have a property name multiply defined, since a property is part of the "state" of an object. Nevertheless, the situation commonly occurs when inheriting from classes defined in libraries or other parts of metamodels. It is then important to provide a way to properly disambiguate property names, i.e. to indicate which actual property is used in an instance.

The situation is worst for operations. MOF provides no semantics for operations, neither for operations themselves, nor for operation inheritance or overriding, because MOF is only concerned with structural definitions. Nevertheless, in the context of Kermeta, one has to provide a clear semantics that does not sacrifice the expressive power of object orientation. Kermeta forbids operation *overloading* (i.e. operations with the same name but different signatures, which is an interesting feature for writing operations whose behaviour depends on how they are called): an operation name is always unique in the context of a class. Furthermore, Kermeta allows a simplified version of overriding, namely *invariant overriding* (i.e. one can redefine the behaviour of an operation in a superclass, but parameter and return types have to be exactly the same) for simplicity reasons. Therefore, the classical technique of dynamic dispatch must be adapted to handle operations in Kermeta.

Resolution techniques for multiple inheritance conflicts have been extensively studied in the object-oriented literature (cf. []). Kermeta choose to "*include a minimal selection mechanism that allows the user to explicitly select the inherited method to override*" [12, §2.9.5.4]: a disambiguation clause introduced by the keyword **from** selects a class among those defining an ambiguous operation to be the one overridden. To follow this design principle and because it is equally needed for property, we extended the **from** clause on properties, resulting in the introduction of the *Feature* class with the *from* reference in Fig. 2.1.

2.1.2 The Kermeta Platform

Kermeta is not only a language, but also a complete platform that eases the creation of DSMLs and their management by proposing several plugins to users. Moreover, Kermeta is involved in several industrial projects, making it mature and robust enough to play a central role in the future of metamodeling. Formally specifying the semantics of the languages at the core of the platform opens the possibility of formal analysis in the future. We roughly overview some of the features of Kermeta platform.

First, concrete syntaxes of DSML can easily be defined and manipulated through the SinTaks plugin. Unfortunately, nothing is available for defining visual concrete syntaxes for now. Second, several metamodeling bridges with commonly used metamodeling formats and tools are available, opening the possibility to Kermeta users to import and export DSLs and models easily and beneficiate from other analysis in other platforms (to name a few, UML class diagrams and statecharts; Marte; and of course, ECore).

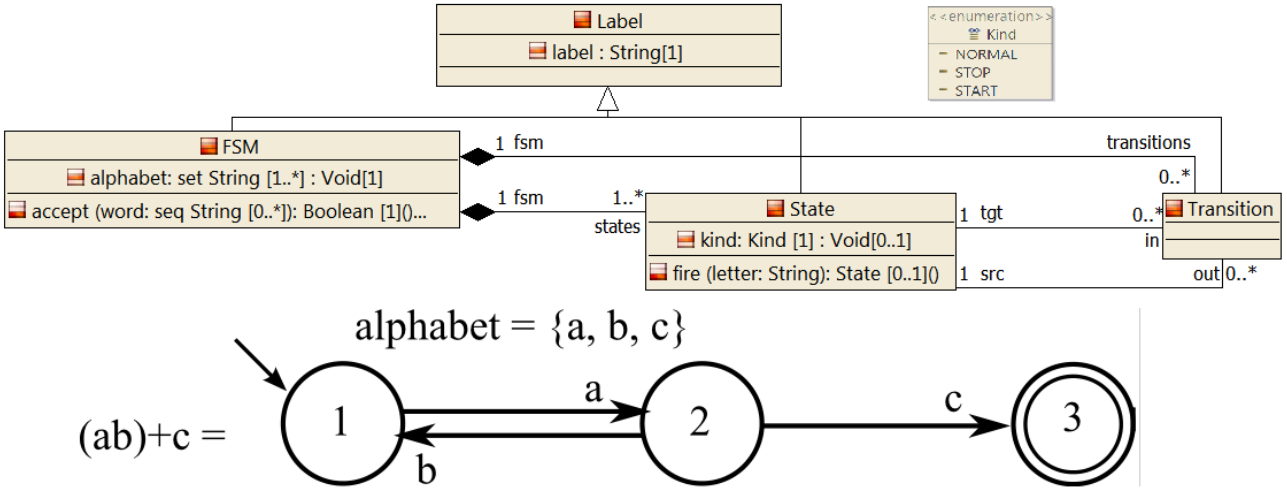


Figure 2.2: The FSM Example: metamodel and model

Kermeta obviously allows one to execute model transformations directly on the platform. Another strength of Kermeta is the possibility to use model composition: build on *KWeaver*, a generic weaver component, Kermeta can deal with heterogeneous and homogeneous model composition (with Kompose and ModMap). This capability simplifies the engineering of models by splitting DSML specifications into several smaller metamodels that are easier to manipulate and understand. Several other metamodeling engineering areas have been explored recently, and although less mature, they are promising: for example, managing feature diagrams, expressing transformations in terms of patern matching of transformation rules, etc.

Kermeta is also involved into several industrial projects, both at the national and european level. To name just a few, Kermeta is involved in the Artist 2 Network of Excellence on Embedded Systems Design, the TopCased Initiative for Critical Applications and Systems Development, AOSD Europe on Aspect Oriented Software Design, and OpenEmbeDD for Real-Time and Embedded Software Design.

2.2 Running Example: The *FSM* Example

A toy example of a Finite State Machine (FSM) is depicted in Fig. 2.2: the FSMs it describes are restricted to possess exactly one initial state and one final state. Unfortunately, these conditions cannot be described through a metamodel, but requires a constraint language like OCL [23], which is beyond the scope of this presentation.

Structurally, the metamodel is included in a package named *FSM*, not visible in the Figure, following following the usual MOF notation (as it encloses the classes). Inside this package, there is four classes *Label*, *FSM*, *State* and *Transition*, and one enumeration *Kind*. The class *FSM* inherits a *label* attribute from *Label*, which represents the FSM's name, and defines a attribute *alphabet* as a (non-empty) set of *String*s, which represents the FSM's possible actions. It contains a (non-empty) sequence of *State*s (through the reference *states*) and a sequence of *Transition*s (through the reference *transitions*) where each element is unique: both inherit a *label* that represents the name of the *State* and the action of the *Transition*. Furthermore, a *State* has a *kind* attribute that determines its nature: either a *START* or a *STOP* (i.e. final) state, or a *NORMAL* state; whereas a *Transition* is attached to a source and a target *State* (corresponding to the *src* and *tgt* references, respectively). Notice that in the usual MOF visual representation of metamodels, the fact that a property is unique or ordered is not directly visible (one has to check the property panel in Eclipse, for example) whereas in the Kermeta editor, it is possible to associate keywords for attributes (cf. the *alphabet* attribute in *FSM*). The Figure 2.2 depicts a conformant FSM with three states and three transitions on the given alphabet (using the classical concrete notation).

The classes *FSM* and *State* declare operations whose body can be defined using the Kermeta's Action Language. In particular, it is possible to *accept* a *word*, given as a sequence of *String*s, that can be recognised by the FSM: if, starting from the *START* state and consuming each *letter* of the *word* by firing the corresponding transition, one finished in a *FINAL* state, the *word* is said to be *accept*ed. The *accept* operation makes use of the *fire* operation in the *State* class: *fire* takes as parameter a *letter* and returns the state reachable from the current one through a transition that carries the same action as the *letter*. The Figure 2.3 gives the Kermeta code for these operations.

```
operation accept(word: seq String [0..*]): Boolean is do
    var current: State init self.getStart()
    var final: State init self.getFinal()
    var toEval: seq String[0..*] init word
    var isNull: Boolean init false
    from var i: Integer init 0
    until i == toEval.size() or isNull loop
        current := current.fire(toEval.at(i))
        if(current.isVoid) then
            isNull := true
        end
        i := i+1
    end
    result := (current == final)
end
```

```
operation fire(letter: String): State [0..1] is do
    var trans: oset Transition [0..*]
        init self.out
    if(trans.isVoid) then
        result := void
    else
        var current: Transition init trans.at(0)
        from var i : Integer init 0
        until i == trans.size() or
            trans.at(i).label == letter loop
            i := i+1
        end
        if(current.isVoid) then
            result:= void
        else
        result:= current.tgt
        end
    end
end
```

Figure 2.3: FSM operations example: on the left, the operation *accept* from class *FSM*; on the right, the operation *fire* from class *State*.

2.3 Mathematical Background

The sign $\triangleq$ defines a set either by extension or by intension. The set of booleans is noted $\mathbb{B} \triangleq \{\top, \bot\}$ for truth values true and false, respectively. The set of naturals and integers are noted $\mathbb{N}$ and $\mathbb{Z}$ respectively; and we define $\mathbb{N}^\star \triangleq \mathbb{N}\backslash\{0\}$; we note $\mathbb{R}$ and $\mathbb{S}$ the sets of real numbers and strings respectively. Let S, S', S'', S''' be sets. The notation $S_\perp$ stands for $S \cup \{\bot\}$, with $S \cap \bot = \varnothing$.

2.3.1 Functions

Since all the mathematical framework is based on set theory and make extensive use of functions, we explicit here the notations used.

Total, resp. partial, functions are noted $f\colon S \to S'$ and $f'\colon S \nrightarrow S'$; their domain is noted $\mathsf{Dom}\,(\cdot)$ and their range $\mathsf{Ran}\,(\cdot)$. Partial functions are right-associative: $g\colon S \nrightarrow S' \nrightarrow S'' \nrightarrow S'''$ means $g\colon S \nrightarrow (S' \nrightarrow (S'' \nrightarrow S'''))$ and we abbreviate $((g(s))(s'))(s'')$ into $g(s)(s')(s'')$, or sometimes write $g_s^{s'}(s'')$ for short. Substituting f by $(s_1, s_2) \in S \times S'$, noted $f[s_1 \mapsto s_2]$, results in a function f' where forall $s \neq s_1$, $f'(s) = f(s)$ and $f'(s_1) = (s_2)$. If $h\colon S \longrightarrow S' \times S''$, we sometimes write $h(s) = (\_, s'')$ or $h(s) = (s', \_)$ when the first or the second element is not relevant in context, and use the same notation for substitution.

2.3.2 Abstract Datatypes Specifications

This Section introduces notations and formal counterparts [**Spivey:1992aa**, 13] for the classical abstract datatypes useful for representing several collections of values, which is a central part of our semantic domains.

Classical collection kinds common in MDE are *bags*, *sets*, *sequences* and *ordered sets*[1] and are considered well-formed, i.e. collections containing values of the same type.

Let C a collection of values of type T, and $t, t' \in \mathsf{T}$ some values. We suppose predefined functions over functions: $\sharp(\mathsf{C})$ returns the number of elements in C and $\mathrm{type}(\mathsf{C})$ returns the type of C, i.e. T. Adding and removing an element t from C is uniformly noted $\mathsf{t} \oplus \mathsf{C}$ and $\mathsf{t} \ominus \mathsf{C}$ respectively, whose effect is properly defined below for each collection kind.

[1]The term usually used in MDE is *ordered set*, which is quite confusing: in set theory, an ordered set is a set equipped with an order *on* the element (i.e. a binary relation); whereas in the MDE vocabulary, an ordered set is a set where the elements are stored, or are accessible, *in* a certain order (which is not a traditional notion of set theory, since it is not usually concerned with computations over sets). This is why in this background, the more precise term "*sequence with unique representative*" is preferred.

Definition 2.1 (Bag Collection)**.** *A* bag *(or* multiset*) of values in* T*, represents a collection where an element can be repeated many times (in contrast to sets) and is noted* $[\mathsf{T}]$.

$$[\mathsf{T}] \triangleq \{\mathsf{T} \nrightarrow \mathbb{N}^\star\}$$

A bag $\{\mathsf{t}_1 \mapsto n_1, \ldots, \mathsf{t}_n \mapsto n_n\}$ is preferably written $[\mathsf{t}_1, \ldots, \mathsf{t}_n]$, where each element t_i appears n_i times in the list $\mathsf{t}_1, \ldots, \mathsf{t}_n$. The empty bag $[\,]$ is a shortcut for the empty function from T to $\mathbb{N}^\star$. Let $\mathsf{B}, \mathsf{B}' \in [\mathsf{T}]$. We note $\mathsf{t} \sharp \mathsf{B}$ the number of times t appears in B; and use the usual set notations for bag membership and subbaging: $\mathsf{t} \in \mathsf{B}$ holds if t appears in B at least one time; and $\mathsf{B} \subseteq \mathsf{B}'$ holds if all elements appearing in B appear in B' at least the same number of times. Adding an element in B, noted $\mathsf{t} \oplus \mathsf{B}$ (or removing it, noted $\mathsf{t} \ominus \mathsf{B}$) changes the number t is repeated, or adds it if it was not there (resp. removes if it appeared only once).

Definition 2.2 (Set Collection)**.** *A set collection is a collection of elements that are not repeated, noted* $\wp(\mathsf{T})$.

Since this notion coincide with the *subset* notion in set theory, it does not need more explanation. The uniform notations for adding $\oplus$ and removing $\ominus$ an element from a set are synonyms of the usual set operators union $\cup$ and difference $\setminus$.

Definition 2.3 (Sequence collections)**.** *A* sequence *(also called* list*) is a collection where the order of the elements matters. We distinguish between sequences allowing multiply repeated values, noted* $\langle \mathsf{T} \rangle$ *and sequences forbidding repetitions, i.e. with a unique representative of each element (also called* ordered sets*), noted* $\langle\!\langle \mathsf{T} \rangle\!\rangle$.

$$\langle \mathsf{T} \rangle \triangleq \{f : \mathbb{N} \nrightarrow \mathsf{T} \mid \mathsf{Dom}\,(f) = \{1..\sharp f\}\}$$
$$\langle\!\langle \mathsf{T} \rangle\!\rangle \triangleq \{f : \mathbb{N} \nrightarrow \mathsf{T} \mid \mathsf{Dom}\,(f) = \{1..\sharp f\} \wedge f \text{ injective}\}$$

A sequence $\{1 \mapsto \mathsf{t}_1, \ldots, n \mapsto \mathsf{t}_n\}$ is preferably noted $\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle$ or $\langle\!\langle \mathsf{t}_1, \ldots, \mathsf{t}_n \rangle\!\rangle$. The empty sequence $\langle\ \rangle$ and $\langle\!\langle\ \rangle\!\rangle$ are shortcuts for the empty function from $\mathbb{N}$ to T. Let $\mathsf{S}, \mathsf{S}' \in \langle \mathsf{T} \rangle$ be two sequences. Adding t on top of S is noted $\mathsf{t} \oplus \mathsf{S}$; removing one occurrence of t in S is noted $\mathsf{t} \ominus \mathsf{S}$. Concatenating two lists, noted $\mathsf{S} \circ \mathsf{S}'$, results in a sequence containing the elements of S followed by those of S'. If S is non-empty, it can be decomposed in $\mathsf{S} = \mathsf{t}_1 :: \mathsf{S}'$, where t is the element at the *head* and S' is the rest, or the *tail*, of the sequence.

Collections possess two orthogonal dimensions: *uniqueness*, i.e. whether a collection admits an element multiply or not; and *ordering*, i.e. whether the order of these elements matter or not. If operations to remove repetition and choose an arbitrary order between unordered elements are available, it is possible to convert any collection to any other one. The following definition introduces two operators that convert collections.

Definition 2.4 (Downward/Upward Conversions)**.** *Let* $\mathsf{Collection} \triangleq \{\mathsf{Bag}, \mathsf{Set}, \mathsf{List}, \mathsf{OSet}\}$. *The family of* downward conversion *operators* $(\bigtriangledown_c)$ *and* upward conversion *operators* $(\bigtriangleup_c)$*, indexed by* $c \in \mathsf{Collection}$ *convert collections adequately by imposing an arbitrary order or removing repetitions of elements.*

For example, let $\mathsf{B} \in [\mathsf{T}]$. Then $\bigtriangledown_{\mathsf{List}}(\mathsf{B})$ results in a list $\mathsf{L} \in \langle \mathsf{T} \rangle$ with all elements of B in an arbitrary order.

Chapter 3

Structural Language

This Chapter formalises the Structural Language (SL) of Kermeta considered in this work: in a nutshell, the SL is a conservative extension of MOF (in its 2.0 version, i.e. without any consideration about genericity) allowing full static typing. The considered extension enables the Action Language to take full advantage of these structures. The formalisation proposed in this Chapter applies directly to any MOF-like language, therefore constituting a canonical semantics for MOF-like structural languages that use the same concepts: packages, enumerations, classes with attributes, references with multiplicities and uniqueness/orderness, and operations.

3.1 Structural Semantics Overview

The Kermeta SL is an conservative extension of the OMG standard MOF. We choose to simplify this language by not considering some of the already existing Kermeta constructions: genericity, model typing and aspects. By isolating this subset of Kermeta's AL, we lose powerful expressive constructions, but gain the fact that the formalisation goes beyond Kermeta: in fact, it works well for any MOF-like SL, even for UML Class Diagrams with the same features.

The MOF Specification [22] comprises two parts: EMOF, (or Essential MOF) and CMOF (or Complete MOF). EMOF is in fact a meta-metamodel: it is a model that describes a language for defining models. CMOF extends EMOF by explicitly defining what the OMG calls the "CMOF *abstract semantics*" [22, §15], i.e. a language (always as a model) that describes what is represented by models.

In Kermeta, things are slightly different. Creating metamodels can be achieved in two ways: either *visually*, by importing a diagrammatic metamodel from another formalism (e.g. UML or ECORE) or by using the dedicated editor; or *textually*, by using the classical Eclipse editor.

For the sake of generality, we explain our formal specification of Kermeta's SL based on the visual representation of EMOF metamodel. The Figure **??** describes our approach. After having described the core artifacts necessary to mathematically specify metamodels and models, we define two sets:

- $\mathcal{M}$ is the set of all Kermeta metamodels that can therefore be seen as a mathematical representation of the EMOF meta-metamodel.
- $\mathbb{M}$ is the set of all Kermeta models by only capturing the syntax of models.

Of course, when a metamodel chooses a particular metamodel $\mathsf{MM} \in \mathcal{M}$, it induces a set of models $\mathsf{M}_1, \ldots, \mathsf{M}_n, \ldots \in \mathbb{M}$ that are *valid* regarding MM (notice that the set of induced models is generally infinite). This notion of validity is traditionally called *conformance* in the MDE community: by defining a model $\mathsf{M} \in \mathbb{M}$, one has to ensure that M is actually one of the models induced by MM, i.e. $\exists i \cdot \mathsf{M} = \mathsf{M}_i$. This relation is noted $\mathsf{M} \blacktriangleright \mathsf{MM}$.

The rest of this Chapter is organised as follows: the next Section defines the core artifacts on which relies all the formalisation: *names*, *types* and *values*. Then, we proceed as explained: Sec. 3.3 formalises MOF metamodels; Sec. 3.4 formalises models; and finally Sec. 3.5 relates models to metamodels by expressing their conformance. The Chapter ends by providing the full representation of the FSM example in terms of our formalisation.

3.2 Names, Types and Values

This Section introduces basic mathematical constructions used all over the formalisation, namely *names*, *(syntactic) types* and *(semantic) values*. These constructions are extracted either from the metamodel itself, providing a basic interpretation of core artifacts in the MOF metamodel, or from the instance model described as a semantic domain for MOF in the MOF Specification Document [22].

3.2.1 Names

In MOF, every concrete class inheriting from the *NamedElement* class should have a name[1]. The following Definition formally defines these concrete elements and associated names.

Definition 3.1 (Elements — Names)**.** *The set* Element *is the set of concrete* MOF *artifacts. The set* Name *represents the names for any* MOF *artifact.*

$$\begin{aligned} \mathsf{Element} &\triangleq \{\mathsf{Pkg}, \mathsf{Class}, \mathsf{Enum}, \mathsf{Attr}, \mathsf{Ref}, \mathsf{Op}, \mathsf{Param}\} \\ \mathsf{Name} &\triangleq (\mathsf{Name}_{\mathsf{e}})_{\mathsf{e} \in \mathsf{Element}} \end{aligned}$$

To avoid the subscripted notation, we introduce a more compact notation for referring to names: for example, the set of class names $\mathsf{Name}_{\mathsf{Class}}$ will be noted ClassN.

Notice that because Name is sorted, it can represent adequately ontologically different artifacts with the same name. In the FSM example, the package named FSM contains a class also named FSM, which is valid in MOF. These is represented as follows: $\mathsf{FSM} \in \mathsf{Name}_{\mathsf{Package}}$ and $\mathsf{FSM} \in \mathsf{Name}_{\mathsf{Class}}$.

Furthermore, we require that packages names in a metamodel are unique, since they are the entry point for accessing metamodel elements. To achieve this requirement, we flatten package names as follows: suppose a package P containing another package PP, then these packages will be represented with the following names: $\mathsf{P}, \mathsf{P} :: \mathsf{P} \in \mathsf{PkgN}$, respectively (or by using any other separator distinct from valid name characters). Consequently, the extraction of names from a metamodel represented as a diagram is straightforward. From now on, unless clearly specified, we will not distinguish between an element and its name in a metamodel.

Finally, the notions of *property* (i.e. either attribute or reference) and *feature* (i.e. either property or operation) names are introduced by following MOF definitions.

$$\begin{aligned} \mathsf{PropN} &\triangleq \mathsf{AttrN} \cup \mathsf{RefN} \\ \mathsf{FeatN} &\triangleq \mathsf{PropN} \cup \mathsf{OpN} \end{aligned}$$

3.2.2 Syntactic Types

MOF defines syntactic types that can be used at the metamodeling level: they correspond to the abstract class *Type* in MOF and Kermeta (cf. Fig. 2.1).

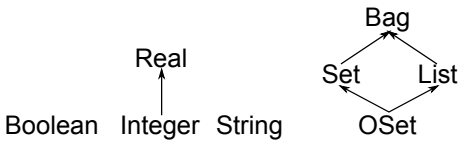
A type (sometimes called *basic* type) is either a *primitive type*, a *class name* or an *enumeration*, both declared in the scope of a given package. Each type is combined with two other information: a *collection kind* and a multiplicity, which constitutes our notion of *syntactic type* MType.

Definition 3.2 (Syntactic Types)**.** *The set of* primitive types *PrimType is one of the classical built-in type name. The set of* (basic) types *Type is either a primitive type, or a class name or an enumeration name declared in the scope of a package. A* collection kind *is one of the usual collection encountered in metamodeling, namely* bag*,* list*,* set *or* ordered set[2]*, or no collection, which is denoted by* $\bot$*. A* collection type *is a pair of a collection and a type. A* multiplicity type *is a pair of a multiplicity, given through its lower and upper bounds, and a collection type. The sets PrimType and Collection are equipped with the usual order (denoted respectively* $\preccurlyeq_{\mathsf{Prim}}$ *and* $\preccurlyeq_{\mathsf{Coll}}$ *represented by the Hasse diagram in the right.*

[1] For a discussion on how MOF actually deals with names, elements identifiers and constraints over names in a metamodel, the reader can refer to Sections §10, §12.4 and §12.5 of the Specification Document [22]. The least we can say is that names and identifiers are intrically defined and should receive proper attention. We choose to simplify the notion of identifier, i.e. how to uniquely refer to a metamodel element, by only considering names. Furthemore, this approach fits well with the way Kermeta's Action Language is defined.

[2] Note here that the traditionally used denomination "ordered set" is confusing: in the MDE area, it denotes a set where the *order of elements* matters, whereas in general it denotes a set equipped with an *order over the elements*.

$$
\begin{aligned}
\mathsf{MType} &\triangleq (\mathbb{N} \times \mathbb{N}_\star) \times \mathsf{CType} \\
\mathsf{CType} &\triangleq \mathsf{Collection} \times \mathsf{Type} \\
\mathsf{Collection} &\triangleq \{\mathsf{Bag}, \mathsf{List}, \mathsf{Set}, \mathsf{OSet}, \bot\} \\
\mathsf{Type} &\triangleq \mathsf{PClassN} \cup \mathsf{DataType} \\
\mathsf{PrimType} &\triangleq \{\mathsf{Boolean}, \mathsf{Integer}, \mathsf{Real}, \mathsf{String}\}
\end{aligned}
$$



In the previous definition, we introduced a shortcut for class or enumeration declared in the scope of a package as following:

$$
\begin{aligned}
\mathsf{DataType} &\triangleq \mathsf{PEnumN} \cup \mathsf{PrimType} \\
\mathsf{PClassN} &\triangleq \mathsf{PkgN} \times \mathsf{ClassN} \\
\mathsf{PEnumN} &\triangleq \mathsf{PkgN} \times \mathsf{EnumN}
\end{aligned}
$$

The set DataType, borrowed from MOF, is constituted by the set of primitive types and the enumerations. Moreover, user-defined types, i.e. enumerations or classes, are always considered with their enclosing package, constituting the sets PClassN and PEnumN: these pairs allow to uniquely refer to either primitive types (unique by nature) or user-defined types.

Let $\mathsf{mt} = ((\mathsf{low}, \mathsf{up}), (\mathsf{C}, \mathsf{t})) \in \mathsf{MType}$ be a multiplicity type. To lighten the notation, we will note $\mathsf{mt} = (\mathsf{low}, \mathsf{up}, \mathsf{C}, \mathsf{t})$. Furthemore, mt is *valid* iff

$$(\mathsf{low} \leqslant \mathsf{up}) \wedge (\mathsf{C} \neq \bot \implies \mathsf{up} > 1)$$

3.2.3 Semantic Values

The set of (semantic) values $\mathbb{V}$ constitutes the core semantic domain for models. It roughly follows the underspecified OMG MOF definitions (cf. [22, §15.2], which is part of CMOF). The definition follows the structural construction of syntactic types, to which they are formally related later in Def. 3.6.

Definition 3.3 ((Semantic) values)**.** *The set* $\mathbb{V}$ *of (semantic) values the (disjoint) union of* basic values sets*, namely the sets of* booleans*,* integers*,* reals*,* strings*,* enumeration literals *and* objects *(also called instances), with the* bags*,* (sub)sets*,* sequences *and* ordered sets *of these basic sets.*

$$
\begin{aligned}
\mathbb{V} \triangleq\ & \mathbb{B} \uplus \mathbb{Z} \uplus \mathbb{R} \uplus \mathbb{S} \uplus \mathbb{E} \uplus \mathbb{O} \\
& [\mathbb{B}] \uplus [\mathbb{Z}] \uplus [\mathbb{R}] \uplus [\mathbb{S}] \uplus [\mathbb{E}] \uplus [\mathbb{O}] \\
& \wp(\mathbb{B}) \uplus \wp(\mathbb{Z}) \uplus \wp(\mathbb{R}) \uplus \wp(\mathbb{S}) \uplus \wp(\mathbb{E}) \uplus \wp(\mathbb{O}) \\
& \langle \mathbb{B} \rangle \uplus \langle \mathbb{Z} \rangle \uplus \langle \mathbb{R} \rangle \uplus \langle \mathbb{S} \rangle \uplus \langle \mathbb{E} \rangle \uplus \langle \mathbb{O} \rangle \\
& \langle\langle \mathbb{B} \rangle\rangle \uplus \langle\langle \mathbb{Z} \rangle\rangle \uplus \langle\langle \mathbb{R} \rangle\rangle \uplus \langle\langle \mathbb{S} \rangle\rangle \uplus \langle\langle \mathbb{E} \rangle\rangle \uplus \langle\langle \mathbb{O} \rangle\rangle
\end{aligned}
$$

This construction, although complicated, does not admit any ill-formed collection of values, i.e. collections with values of different sets (for example, a list with one integer and one boolean). The *size* of a value $|\bullet| : \mathbb{V} \to \mathbb{N}$ is defined by extension from Sec. 2.3.2. The type of a value will be defined later, when formal definitions of metamodels and models will be available.

3.3 Metamodels

A metamodel consists basically in declarations: MOF elements (through their name) are bind to (syntactic) types and other information, w.r.t. a particular topology defined by MOF. This Section defines several sets of functions capturing the structure of these declarations. Of course, all these declarations make sense when put together to define the metamodel itself.

This Section proceeds as follows: for each MOF artifact, an informal description based on the metamodel representation in Fig. 2.1 is provided, from which a formal definition is derived by providing adequate mathematical structures in order to create a set of functions formalising the idea of this artifact. As we said before, the formalisation only addresses the *concrete* classes of the meta-metamodel (namely, the classes corresponding to *packages*, *enumerations*, *classes*, *properties* and *operations*), which are actually instanciated in a particular metamodel specification.

3.3.1 Package

From the metamodel in Fig. 2.1, the class *Package* has two references: *subpackages* (with its opposite *superpackage*) allows a package to eventually contain subpackages; and *types* (with its opposite *package*) allows a package to eventually contains types, i.e. either enumerations or classes.

The set $\mathcal{P}$ of *package functions* represents the *packages declarations*: such a function $p \in \mathcal{P}$ maps packages that are part of a metamodel to their subpackages, and nested classes and enumerations; and p is *correct* if no package is contained in itself.

$$\mathcal{P} \triangleq \{p : \mathsf{PkgN} \nrightarrow \wp(\mathsf{PkgN}) \times \wp(\mathsf{ClassN}) \times \wp(\mathsf{EnumN}) \mid \forall \mathsf{pkg}, p(\mathsf{pkg}) = (P, C, E) \Rightarrow \mathsf{pkg} \notin P\}$$

Notice here why we required that package names need to be unique throughout a metamodel: $\mathcal{P}$, as well as the following definitions, is a function from package names, which implies to be able to uniquely access any metamodel package through its name. This is generally not true in MOF: it is valid in MOF to have a package named P containing a subpackage of the same name P. The package name uniqueness requirement does not change the structural consistency of metamodels, and the convention we proposed is not ambiguous, allowing one to retrieve the original metamodel names.

3.3.2 Enumeration

From Fig. 2.1, the class *Enum* has one reference *literals* (with its opposite reference *enum*) to the class *EnumLit*, constraining an enumeration to possess at least one literal. The *EnumLit* class also inherits from *NamedElement*, making any literal possessing a name, and the set of literals for an enumeration is ordered.

The set $\mathcal{E}$ of *enumeration functions* represents *enumerations declarations*: such a function $e \in \mathcal{E}$ maps enumerations declared inside a package to an ordered set of enumeration literals.

$$\mathcal{E} \triangleq \{e : \mathsf{PkgN} \nrightarrow \mathsf{EnumN} \nrightarrow \langle\!\langle \mathbb{E} \rangle\!\rangle\}$$

Again, we impose that enumeration literals are unique throughout a metamodel. This is also generally not true, neither in MOF or in Kermeta. Nevertheless, this constraint seems reasonable: in Kermeta, enumeration literals are always qualified by their enumeration name (cf. Appendix A in *getStart*'s body). Like for package names, it is always possible to ensure this requirement by considering qualified enumeration literal names.

Under this condition, it is possible to define a reverse function $enum$ that retrieves the enumeration which a literal is defined in.

$$\begin{array}{rcl} enum : \mathsf{EnumLit} & \rightarrow & \mathsf{PkgN} \times \mathsf{Enum} \\ \mathsf{lit} & \mapsto & (\mathsf{pkg}, \mathsf{enum}) \text{ if } \mathsf{lit} \in e(\mathsf{pkg})(\mathsf{enum}) \end{array}$$

3.3.3 Class

From Fig. 2.1, the class *Class* has one attribute and two references. The attribute *abstract* indicates that a class is abstract; it implies that the class is concretely represented with its name in italic, and make this class not directly instanciable. The *supers* reference indicates that a class can refer to superclasses from which it inherits; and the *features* reference (with its opposite *class*) allows a class to declare features, i.e. properties and/or operations.

The set $\mathcal{C}$ of *class functions* represents *classes declarations*: such a function $c \in \mathcal{C}$ maps classes declared inside a package to a boolean indicating if they are abstract, and the set of their superclasses; it is *correct* if no class inherits from itself, and if the inheritance hierarchy is acyclic.

$$\mathcal{C} \triangleq \{c : \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathbb{B} \times \wp(\mathsf{ClassN}) \mid (\mathsf{pkg}, \mathsf{c}) \not\prec^{+}_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c})\}$$

The previous definition uses an order relation $\prec_{\mathsf{Class}} \subseteq \mathsf{PClassN} \times \mathsf{PClassN}$, induced by the inheritance hierarchy inside the same package (i.e. $(\mathsf{pkg}, \mathsf{c}) \prec_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c}')$ if c defines c' as one of its superclass). The relation $\prec^{+}_{\mathsf{Class}}$ used in the previous definition is the transitive closure of this order.

$$\forall \mathsf{pkg} \in \mathsf{Dom}\,(c), (\mathsf{pkg}, \mathsf{c}) \prec_{\mathsf{Class}} (\mathsf{pkg}, \mathsf{c}') \overset{\triangle}{\Longleftrightarrow} \mathsf{c}, \mathsf{c}' \in \mathsf{Dom}\,(c(\mathsf{pkg})) \ \wedge \ c(\mathsf{pkg})(\mathsf{c}) = (\_, C) \wedge \mathsf{c}' \in C\}$$

This order relation is then extended to cover primitive types: we define the order relation $\preccurlyeq_{\mathsf{Type}} \subseteq \mathsf{Type} \times \mathsf{Type}$ by combining the order relations on each type sort (enumerations are not ordered); and then the order relation $\preccurlyeq \subseteq \mathsf{CType} \times \mathsf{CType}$ to cover collection type:

$$\begin{array}{rcl} \preccurlyeq_{\mathsf{Type}} & = & \preccurlyeq_{\mathsf{Class}} \cup \preccurlyeq_{\mathsf{Prim}} \\ (\mathsf{C},\mathsf{t}) \preccurlyeq (\mathsf{C}',\mathsf{t}') & \stackrel{\triangle}{\iff} & \mathsf{C} \preccurlyeq_{\mathsf{Coll}} \mathsf{C}' \wedge \mathsf{t} \preccurlyeq_{\mathsf{Type}} \mathsf{t}' \end{array}$$

3.3.4 Property

In Fig. 2.1, the class *Property* inherits from *Feature*, which in turn inherits from *CollectionType*. The class *Feature* has one reference *from* that indicates optionally indicates which class should be considered if the feature (name) is multiply inherited, and a *class* reference indicating that a feature is always always declared, or contained, in one class. The class *CollectionType* simply represents an MType. Since *Property* is abstract, we should look at the concrete classes that inherit from it: the *Attribute* class has no properties, but is required to possess a *DataType* as a *type*; whereas the *Reference* class is required to possess a *Class* as a *type* (cf. [33]). Furthemore, the *Reference* class has one attribute *containment*, indicating if the reference is a containment (which fact is notationally represented by a black diamond in the concrete notation); and an *opposite* reference indicating that a reference admits another reference from its type as an opposite (which is notationally represented by removing the arrow head).

The set $\mathcal{P}\mathsf{rop}$ of *property functions* represents *properties declarations*: such a function $\mathrm{prop} \in \mathcal{P}\mathsf{rop}$ maps each class property to a boolean indicating if it is contained, an optional class used for the from clause, the multiplicity type of the property and its optional opposite reference.

$$\mathcal{P}\mathsf{rop} \stackrel{\triangle}{=} \{\mathrm{prop} : \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{PropN} \nrightarrow \mathbb{B} \times \mathsf{ClassN}_\perp \times \mathsf{MType} \times \mathsf{RefN}_\perp\}$$

Suppose that $\mathrm{prop}(\mathsf{pkg})(\mathsf{class})(\mathsf{p}) = (cnt, \mathsf{from}, \mathsf{mt}, \mathsf{opp})$ with $\mathsf{mt} = (low, up, \mathsf{coll}, \mathsf{t})$ is a property function. Then prop is *correct* if it respects the previously stated constraints:

- if $\mathsf{p} \in \mathsf{AttN}$ is an attribute, it is always contained, does not possess an opposite, and its type is DataType.

$$cnt = \top \wedge \mathsf{t} \in \mathsf{DataType} \wedge \mathsf{opp} = \perp$$

- if $\mathsf{p} \in \mathsf{RefN}$ is a reference, then its type is a class and its opposite reference opp is conversly mapped to the enclosing class of p

$$\mathsf{t} = (\mathsf{pkg}', \mathsf{c}') \in \mathsf{PClassN} \wedge \mathsf{opp} \neq \perp \implies \left\{ \begin{array}{rcl} \mathrm{prop}(\mathsf{pkg}')(\mathsf{c}')(\mathsf{opp}) & = & (cnt', \mathsf{from}', \mathsf{mt}', \mathsf{p}) \\ \text{with } \mathsf{mt}' & = & (low', up', \mathsf{coll}', \mathsf{pkg}, \mathsf{c}) \\ \text{and } cnt = \top & \implies & cnt' = \perp \wedge (low', up') = (0,1) \end{array} \right.$$

3.3.5 Operation

In Fig. 2.1, the class *Operation* inherits from *Feature*, which in turn inherits from *CollectionType*. Therefore, it shares with the *Property* class the same capabilities. Furthemore, the *Operation* class has one attribute and two references. The *abstract* attribute indicates that an operation is abstract, which means that it has no body and is also contained in an abstract class. The *parameters* reference points to the *Parameter* class, which represents the (ordered) list of parameters, which are typed because the *Parameter* class inherits from *CollectionType*. The *statements* reference points to the *Body* class and corresponds to the body of the operation, which precise definition is the purpose of Chapter 4.

The set $\mathcal{O}$ of *operation functions* represents *operations declarations*: such a function $o \in \mathcal{O}$ maps each class operation to a boolean indicating if it is abstract, an optional class used for the from clause, a multiplicity type representing the return type (where $\perp$ corresponds to `void`), a (ordered) sequence of parameters together with their types, and the definition of the body; and o is correct if every operation defined as abstract does not contain a body.

$$\begin{array}{rcl} \mathcal{O} & \stackrel{\triangle}{=} & \{o\colon \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{OpN} \nrightarrow \mathbb{B} \times \mathsf{ClassN}_\perp \times \langle\!\langle \mathcal{P}\mathsf{arams} \rangle\!\rangle \times \mathsf{MType}_\perp \times \mathsf{Body}_\perp \mid \\ & & \forall \mathsf{pkg} \in \mathsf{PkgN}, \mathsf{c} \in \mathsf{ClassN}, \mathsf{op} \in \mathsf{OpN}, o(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) = (abs, \_, \_, \_, b) \cdot abs \implies b = \perp\} \\ \mathcal{P}\mathsf{arams} & \stackrel{\triangle}{=} & \mathsf{ParamN} \times \mathsf{MType} \end{array}$$

Here, $\mathcal{P}\mathsf{arams}$ is defined as a pair containing the parameter name and its multiplicity type.

3.3.6 Metamodel

The set of metamodel $\mathcal{M}$ can now be defined by putting the previous definitions together. Note that the use of (partial) function naturally ensures Kermeta's constraint on unicity of names: it is not possible to represent, in our framework, e.g. a metamodel that would contain two packages (or two properties in the same class) with the same name.

Definition 3.4 (Metamodel $\mathcal{M}$)**.** *A* metamodel $\mathsf{MM} \in \mathcal{M}$ *is a tuple* $\mathsf{MM} = (p, c, e, \mathrm{prop}, o) \in \mathcal{P} \times \mathcal{C} \times \mathcal{E} \times \mathcal{P}\mathsf{rop} \times \mathcal{O}$.

Let $\mathsf{MM} = (p, c, e, \mathrm{prop}, o) \in \mathcal{M}$ be a metamodel of interest. The set of *qualified property names* QPropN (respectively, *qualified operation names*, and *qualified feature names*) is the set of property (resp. operation, feature) names correctly defined in the scope of MM:

$$\mathsf{QPropN}_{\mathsf{MM}} \triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{p}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{PropN} \mid \mathsf{pkg} \in \mathrm{Dom}\,(\mathrm{prop}) \wedge \mathsf{c} \in \mathrm{Dom}\,(\mathrm{prop}(\mathsf{pkg})) \wedge \mathsf{p} \in \mathrm{Dom}\,(\mathrm{prop}(\mathsf{pkg})(\mathsf{c}))\}$$
$$\mathsf{QOpN}_{\mathsf{MM}} \triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{OpN} \mid \mathsf{pkg} \in \mathrm{Dom}\,(o) \wedge \mathsf{c} \in \mathrm{Dom}\,(o(\mathsf{pkg})) \wedge \mathsf{p} \in \mathrm{Dom}\,(o(\mathsf{pkg})(\mathsf{c}))\}$$
$$\mathsf{QFeatureN}_{\mathsf{MM}} \triangleq \{(\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{PkgN} \times \mathsf{ClassN} \times \mathsf{FeatureN} \mid (\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{QPropN} \vee (\mathsf{pkg}, \mathsf{c}, \mathsf{f}) \in \mathsf{QOpN}\}$$

A number of functions are also defined to ease the manipulation of a particular metamodel's information. These functions act like projector functions, i.e. they select a particular element in the image of the functions constituting the metamodel.

$$\begin{array}{rcl} super_{\mathsf{MM}} & : & \mathsf{PClassN} \longrightarrow \wp(\mathsf{Class}) \\ \mathrm{abs}_{\mathsf{MM}} & : & \mathsf{QOpN} \cup \mathsf{PClassN} \longrightarrow \mathbb{B} \\ \mathrm{partypes}_{\mathsf{MM}} & : & \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{CType} \rangle\!\rangle \end{array} \qquad \begin{array}{rcl} \mathrm{from}_{\mathsf{MM}} & : & \mathsf{QFeatN} \longrightarrow \mathsf{Class}_{\perp} \\ \mathrm{type}_{\mathsf{MM}} & : & \mathsf{QFeatN} \longrightarrow \mathsf{MType}_{\perp} \\ \mathrm{parnames}_{\mathsf{MM}} & : & \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{ParamN} \rangle\!\rangle \end{array}$$

We give only the formal definition for super, the other functions are built the same way from MM's functions. Let $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$ such that $\mathsf{c} \in \mathrm{Dom}\,(c(\mathsf{pkg}))$, $\mathrm{super}_{\mathsf{MM}}(\mathsf{pkg}, \mathsf{c}) = C \overset{\triangle}{\Longleftrightarrow} c_{\mathsf{MM}}(\mathsf{pkg})(c) = (\_, C)$: it represents the set of $(\mathsf{pkg}, \mathsf{c})$ superclasses. Similarly, abs indicates if a class $(\mathsf{pkg}, \mathsf{c})$ or an operation $(\mathsf{pkg}, \mathsf{c}, \mathsf{op})$ is abstract; from retrieves the disambiguation class (if any) of a feature; and type the multiplicity type of a feature (which can be $\perp$, i.e. *void*, in the case of an operation); and partypes and parnames retrieve the ordered list of respectively the collection types and the names of an operation's parameters.

A metamodel $\mathsf{MM} = (p, e, c, \mathrm{prop}, o)$ is *valid* if all classes declared inside a package also appear in the domains of c, prop and o under the same package, and every class containing an abstract operation is also declared abstract[3].

$$p(\mathsf{pkg}) = (P, C, E) \implies \begin{cases} E = \mathrm{Dom}\,(e(\mathsf{pkg})) \\ C = \mathrm{Dom}\,(c(\mathsf{pkg})) \\ C = \mathrm{Dom}\,(\mathrm{prop}(\mathsf{pkg})) \\ C = \mathrm{Dom}\,(o(\mathsf{pkg})) \end{cases}$$
$$abs(\mathsf{pkg}, \mathsf{c}, \mathsf{o}) \implies abs(\mathsf{pkg}, \mathsf{c})$$

3.4 Models

A metamodel defines a set of models. A model consists basically in a collection of *objects* (or equivalently called *instances*). Each object has a *type*, i.e. a class of a metamodel, and maintains a *state*. An object's state is intuitively a mapping between property names and values, in such a way that the involved names correspond to the declared names for the object's type.

3.4.1 Accessible Features

In the presence of inheritance, and especially multiple inheritance, referring to features by their names might be ambiguous because the names can be repeated throughout the inheritance hierarchy. In fact, we will show that for a given object, all property name is unique with respect to the disambiguation clause.

Suppose now a model $\mathsf{M} \in \mathbb{M}$ of a metamodel $\mathsf{MM} \in \mathcal{M}$, and an object o of type $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$. To be correctly defined, o's state should associate a value to all *accessible* property declared in MM, i.e. these properties defined in the

[3] Strictly speaking, this constraint comes from Kermeta: "*Kermeta requires that every class that contains an abstract operation must be declared as an abstract class.*" [12, §2.8.2]

inheritance scope of c: either directly declared in c, or inherited from c's superclasses. Similarly, o's accessible operations are these operations that are either defined directly in c or inherited from superclasses.

We define two functions to capture *accessible features*: the function π_{MM} over a metamodel MM (omitted when clear from context) recursively computes the information of all properties accessible from classes of MM; and the function ω_{MM} recursively computes the information of all accessible operations.

$$\begin{array}{rcl} \pi_{\mathsf{MM}} & : & \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{PropN} \nrightarrow \mathbb{B} \times \mathsf{MType} \times \mathsf{RefN}_{\perp} \\ \omega_{\mathsf{MM}} & : & \mathsf{PkgN} \nrightarrow \mathsf{ClassN} \nrightarrow \mathsf{OpN} \nrightarrow \mathbb{B} \times \langle\!\langle \mathcal{P}\mathsf{arams} \rangle\!\rangle \times \mathsf{MType}_{\perp} \times \mathsf{Body} \end{array}$$

Notice that the signature of these functions is slightly different: since they compute the accessible features, they are not ambiguous any more: no component records the disambiguation clause.

Functions π_{MM} and ω_{MM} are well-founded because they follow the well-founded order $\prec_{\mathsf{Class}}$ over classes. Nevertheless, we must ensure that they are well-defined, i.e. *each accessible property for o has an unique name* and *each call resolves into a unique operation*. Why these two properties hold come from different reasons; we only sketch the proof for each of them.

Properties. First, as we mentioned before, direct property names are unique. Second, Kermeta, following MOF, prevents from the possibility of property overriding because "*it simply does not make sense from a structural point of view*" [12, §2.9.5]: this ensures that in a single inheritance path, property names are unique. Finally, any class with multiple superclasses must ensure uniqueness of property names by using the *from* disambiguation clause.

Operations. First, as we mentioned before, overloading is forbidden in Kermeta: inside a given class, operation names are unique. Second, Kermeta allows invariant overriding, meaning that in a single inheritance path, an operation call resolves to the closest operation up to the class hierarchy. Finally, in case of multiply inherited operations, each class must ensure the uniqueness of the inherited operations [12, §2.9.5.4] by using the *from* disambiguation clause.

Consequently, each accessible property has a unique name and we can define a function decl that associate *the* class where a property name is defined.

$$\begin{array}{rcl} \mathrm{decl}\colon \mathsf{PClassN} \times \mathsf{PropN} & \nrightarrow & \mathsf{PClassN} \\ ((\mathsf{pkg}, \mathsf{c}), \mathsf{propN}) & \mapsto & \begin{cases} (\mathsf{pkg}, \mathsf{c}') & \text{if } \mathrm{from}(\mathsf{pkg}, \mathsf{c}, \mathsf{propN}) = \perp \\ & \text{and } \mathsf{propN} \in \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}')) \\ (\mathsf{pkg}, \mathsf{c}'') & \text{if } \mathrm{from}(\mathsf{pkg}, \mathsf{c}, \mathsf{propN}) = \mathsf{c}' \\ & \text{and } \mathsf{propN} \in \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}')) \end{cases} \end{array}$$

As operations rely on dynamic lookup, i.e. the chosen operation's body for a call depends on the dynamic type of the object, it is not possible to define a similar function. The decision will be made dynamically (cf. 4.4.3).

3.4.2 Model

The following definition only translates in a mathematical structure the considerations made at the beginning of this Section, taking advantage of the previous remarks on uniqueness of properties.

Definition 3.5 (Model $\mathbb{M}$)**.** *The set of models $\mathbb{M}$ is a set of functions that associate model objects to their type and state.*

$$\begin{array}{rcl} \mathbb{S}\mathsf{tate} & \triangleq & \{\sigma : \mathsf{PropN} \nrightarrow \mathbb{V}\} \\ \mathbb{M} & \triangleq & \{\mathsf{M} : \mathbb{O} \nrightarrow \mathsf{PClassN} \times \mathbb{S}\mathsf{tate}\} \end{array}$$

Given a model $\mathsf{M} \in \mathbb{M}$, we note σ^o_{M} the state of the object $o \in \mathbb{O}$, if $o \in \mathsf{Dom}\,(\mathsf{M})$ and $type_{\mathsf{M}}(o)$ its type (subscripts are omitted if clear from context).

The following definition relates values to (collection) types in the context of a model: types of scalar values (booleans, integers, reals and strings) do not depend on the model; the type of an enumeration literal is the enumeration where this literal is defined; and the type of an object is the object's type in the model.

Definition 3.6 (Type of a value)**.** *The function τ trivially associates a (syntactic) type to a (semantic) value.*

$$\tau_{\mathsf{M},\mathsf{MM}} : \mathbb{V} \longrightarrow \mathsf{CType}$$
$$v \mapsto \begin{cases} (\bot, \mathsf{Boolean}) & \text{if } v \in \mathbb{B} \\ (\bot, \mathsf{Integer}) & \text{if } v \in \mathbb{N} \\ (\bot, \mathsf{Real}) & \text{if } v \in \mathbb{R} \\ (\bot, \mathsf{String}) & \text{if } v \in \mathbb{S} \\ (\bot, \mathrm{enum}_{\mathsf{MM}}(v)) & \text{if } v \in \mathbb{E} \\ (\bot, \mathrm{type}_{M}(v)) & \text{if } v \in \mathbb{O} \\ \dots & \\ (\mathsf{Bag}, \mathsf{Boolean}) & \text{if } v \in [\mathbb{B}] \\ (\mathsf{Set}, \mathsf{Boolean}) & \text{if } v \in \wp(\mathbb{B}) \\ \dots & \end{cases}$$

3.5 Conformance

We say that M *conforms to* MM, and note $\mathsf{M} \blacktriangleright \mathsf{MM}$, if M actually belongs to the set induced by MM. This predicate is defined recursively: $\mathsf{M} \blacktriangleright \mathsf{MM}$ holds if all objects of the metamodel respect these conditions: $\forall o \in \mathsf{Dom}\,(\mathsf{M})$,

- o's type is declared in MM and all accessible property from this type has a value;

$$\mathrm{type}(o) = (\mathsf{pkg}, \mathsf{c}) \Longrightarrow \begin{cases} \mathsf{p} \in \mathsf{Dom}\,(p) \\ p(\mathsf{pkg}) = (P, C, E) \Rightarrow \mathsf{c} \in C \\ (\mathsf{Dom}\,(\sigma^o) = \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c})) \\ \forall \mathsf{p} \in \mathsf{Dom}\,(\sigma^o), \sigma^o(\mathsf{p}) \in \mathbb{V} \end{cases}$$

- For each property $\mathsf{p} \in \mathsf{PropN}$ that is accessible from o (i.e. $\mathsf{p} \in \mathsf{Dom}\,(\sigma^o)$ and that has value $v \in \mathbb{V}$ and type $\mathsf{mt} = (low, up, \mathsf{C}, \mathsf{t}) \in \mathsf{MType}$ (i.e. $\sigma^o(\mathsf{p}) = v$ and $\pi(\mathsf{pkg})(\mathsf{c})(\mathsf{p}) = \mathsf{mt}$), then (i) v's type specialises ct; and (ii) v's size respects p's declared bounds.

$$\begin{array}{llll} \text{(i)} & \mathrm{type}(v) & \preccurlyeq & (\mathsf{C}, \mathsf{t}) \\ \text{(ii)} & low \leqslant & \mid v \mid & \leqslant up \end{array}$$

- Furthemore, if v is a collection of (internal) values $v_1, \dots, v_n$ (i.e. $\mathsf{C} \neq \bot$) then each of these v_i has a type that specialises t (remember that collection values are always well-formed, cf. Def. 3.3;

$$\forall i, type(v_i) \preccurlyeq \mathsf{t}$$

- Finally, if the property p is a reference with an opposite property opp (i.e. $\mathsf{p} \in \mathsf{RefN}$ such that $\mathrm{opp}(\mathsf{pkg}, \mathsf{c}, \mathsf{p}) = \mathsf{opp} \neq \bot$), then opp's value is either an object v itself, or a collection of (internal) objects $v'_1 \dots v'_n$; in this case, the original object o must be included between opp's (internal) value(s).

$$\mathsf{opp} \neq \bot \Longrightarrow \forall i, o \subseteq \sigma^{v_i}(\mathsf{opp})$$

3.6 Example

This last Section explains how the example presented in Fig. 2.2 fits our formalisation, by writing in terms of the mathematical structures presented in this Chapter.

3.6.1 The FSM Metamodel $\mathsf{MM}_{\mathsf{FSM}}$

Let us call $\mathsf{MM}_{\mathsf{FSM}} = (p, c, e, \mathrm{prop}, o) \in \mathcal{M}$ the mathematical representation of the metamodel depicted in Fig. 2.2. Because the mathematical framework uses partial functions, we have to specify the functions' domains first: $\mathsf{Dom}\,(p) = \mathsf{Dom}\,(c) = \mathsf{Dom}\,(\mathrm{prop}) = \mathsf{Dom}\,(o) = \{\mathsf{FSM}\}$ (only one package); $\mathsf{Dom}\,(c(\mathsf{FSM})) = \mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})) = \mathsf{Dom}\,(o(\mathsf{FSM})) = \mathsf{C}_{\mathsf{FSM}}$ with $\mathsf{C}_{\mathsf{FSM}} = \{\mathsf{Label}, \mathsf{FSM}, \mathsf{State}, \mathsf{Transition}\}$, i.e. the classes contained inside the FSM package; and $\mathsf{Dom}\,(e(\mathsf{FSM})) =$

$\mathsf{E}_{\mathsf{FSM}}$ with $\mathsf{E}_{\mathsf{FSM}} = \{\mathsf{Kind}\}$. The domains for prop and o are build by gathering all properties and operations respectively: $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})) = \{\mathsf{label}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{Label})) = \varnothing$; $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{alphabet}, \mathsf{states}, \mathsf{transitions}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{getStart}, \mathsf{getFinal}, \mathsf{accept}\}$; $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{fsm}, \mathsf{kind}, \mathsf{in}, \mathsf{out}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{fire}\}$; and finally $\mathsf{Dom}\,(\mathrm{prop}(\mathsf{FSM})\ (\mathsf{Transition})) = \{\mathsf{fsm}, \mathsf{src}, \mathsf{tgt}\}$ and $\mathsf{Dom}\,(o(\mathsf{FSM})(\mathsf{Transition})) = \varnothing$.

We then have to define all the component functions of $\mathsf{MM}_{\mathsf{FSM}}$. Let us start with the simplest ones, namely the package, class and enumeration functions.

$$\begin{aligned}
p(\mathsf{FSM}) &= (\varnothing, \mathsf{C}_{\mathsf{FSM}}, \mathsf{E}_{\mathsf{FSM}}) \\
e(\mathsf{FSM})(\mathsf{Kind}) &= \langle\!\langle \mathsf{NORMAL}, \mathsf{STOP}, \mathsf{START} \rangle\!\rangle \\
c(\mathsf{FSM})(\mathsf{Label}) &= (\top, \varnothing) \\
c(\mathsf{FSM})(\mathsf{FSM}) &= (\bot, \{\mathsf{Label}\}) \\
c(\mathsf{FSM})(\mathsf{State}) &= (\bot, \{\mathsf{Label}\}) \\
c(\mathsf{FSM})(\mathsf{Transition}) &= (\bot, \{\mathsf{Label}\})
\end{aligned}$$

Let us now proceed with the property and operation functions, class by class.

$$\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$$

$$\begin{aligned}
\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{alphabet}) &= (\top, \bot, (1, \star, \mathsf{Set}, \mathsf{String}), \bot) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{states}) &= (\top, \bot, (1, \star, \mathsf{OSet}, \mathsf{State}), \mathsf{fsm}) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) &= (\top, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm}) \\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{accept}) &= (\bot, \bot, \langle\!\langle (\mathsf{word}, (0, \star, \mathsf{List}, \mathsf{String})) \rangle\!\rangle, (1, 1, \bot, \mathsf{Boolean}), b_{\mathsf{accept}}) \\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getStart}) &= (\bot, \bot, \langle\!\langle\rangle\!\rangle, (1, 1, \bot, \mathsf{State}), b_{\mathsf{getStart}}) \\
o(\mathsf{FSM})(\mathsf{FSM})(\mathsf{getFinal}) &= (\bot, \bot, \langle\!\langle\rangle\!\rangle, (1, 1, \bot, \mathsf{State}), b_{\mathsf{getFinal}})
\end{aligned}$$

$$\begin{aligned}
\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{kind}) &= (\top, \bot, (1, 1, \bot, \mathsf{Kind}), \bot) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{fsm}) &= (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{states}) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) &= (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt}) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{out}) &= (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{src}) \\
o(\mathsf{FSM})(\mathsf{State})(\mathsf{fire}) &= (\bot, \bot, \langle\!\langle (\mathsf{letter}, (1, 1, \bot, \mathsf{String})) \rangle\!\rangle, (0, 1, \bot, \mathsf{State}), b_{\mathsf{fire}})
\end{aligned}$$

$$\begin{aligned}
\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) &= (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions}) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{src}) &= (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{out}) \\
\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) &= (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})
\end{aligned}$$

3.6.2 The FSM Model $\mathsf{M}_{\mathsf{abc}}$

Let us now call $\mathsf{M}_{\mathsf{abc}} \in \mathbb{M}$ the representation of the model depicted in Fig. 2.2. Using the names as object identifiers, we have $\mathsf{Dom}\,(\mathsf{M}_{\mathsf{abc}}) = \{abc, 1, 2, 3, a, b, c\}$. We obviously have $type(abc) = (\mathsf{FSM}, \mathsf{FSM})$, $type(1) = type(2) = type(3) = (\mathsf{FSM}, \mathsf{State})$ and $type(a) = type(b) = type(c) = (\mathsf{FSM}, \mathsf{Transition})$. We only describe the state of the necessary instances for the conformance proof.

$$\begin{aligned}
\sigma^{abc}(\mathsf{label}) &= \text{"(ab) + c"} \\
\sigma^{abc}(\mathsf{alphabet}) &= \{\text{"a"}, \text{"b"}, \text{"c"}\} \\
\sigma^{abc}(\mathsf{states}) &= \langle\!\langle 1, 2, 3 \rangle\!\rangle \\
\sigma^{abc}(\mathsf{transitions}) &= \langle\!\langle a, b, c \rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
\sigma^{a}(\mathsf{label}) &= \text{"a"} \\
\sigma^{a}(\mathsf{src}) &= 1 \\
\sigma^{a}(\mathsf{tgt}) &= 2 \\
\sigma^{a}(\mathsf{fsm}) &= abc
\end{aligned}$$

$$\begin{aligned}
\sigma^{1}(\mathsf{label}) &= \text{"1"} \\
\sigma^{1}(\mathsf{kind}) &= \mathsf{START} \\
\sigma^{1}(\mathsf{in}) &= \langle\!\langle b \rangle\!\rangle \\
\sigma^{1}(\mathsf{out}) &= \langle\!\langle a \rangle\!\rangle \\
\sigma^{1}(\mathsf{fsm}) &= abc
\end{aligned}$$

$$\begin{aligned}
\sigma^{b}(\mathsf{label}) &= \text{"b"} \\
\sigma^{b}(\mathsf{src}) &= 2 \\
\sigma^{b}(\mathsf{tgt}) &= 1 \\
\sigma^{b}(\mathsf{fsm}) &= abc
\end{aligned}$$

3.6.3 Does $\mathsf{M}_{\mathsf{abc}}$ conform to $\mathsf{MM}_{\mathsf{FSM}}$?

We now prove the conformance, i.e. $\mathsf{M}_{\mathsf{abc}} \blacktriangleright \mathsf{MM}_{\mathsf{FSM}}$. The conformance predicate holds if certain conditions hold on all objects of the model. Since it is a repetitive task, we only demonstrate for one object of each type, letting the reader infer what is missing for the rest of the objects.

The first condition is easy to check from the definition of $\mathsf{M_{abc}}$ itself: each object has a type that appears in $\mathsf{MM_{FSM}}$, and each accessible property of each object possess a value.

It then remains to prove that each accessible property of each object has a valid value regarding the property declared type, and so do opposites. We only select one object per type, i.e. the object already presented when describing $\mathsf{M_{abc}}$.

FSM An FSM instance has four accessible properties: $\mathrm{Dom}\,(\pi(\mathsf{FSM})(\mathsf{FSM})) = \{\mathsf{label}, \mathsf{alphabet}, \mathsf{states}, \mathsf{transitions}\}$. We go through all of them, since they all have different types:

label From the previous definitions, we have $\sigma^{abc}(\mathsf{label}) = \text{"}(\mathsf{ab}) + \mathsf{c}\text{"}$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}),$

- Since $\mathrm{type}(\text{"}(\mathsf{ab}) + \mathsf{c}\text{"}) = (\bot, \mathsf{String})$, it implies that $\mathrm{type}(\text{"1"}) \preccurlyeq (\bot, \mathsf{String})$;
- Since $|\ \text{"}(\mathsf{ab}) + \mathsf{c}\text{"}\ | = 1$, it implies that $1 \leqslant |\ \text{"}(\mathsf{ab}) + \mathsf{c}\text{"}\ | \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

alphabet We have $\sigma^{abc}(\mathsf{alphabet}) = \{\text{"a"}, \text{"b"}, \text{"c"}\}$ with $\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{alphabet}) = (\top, \bot, (1, \star, \mathsf{Set}, \mathsf{String}), \bot)$.

- Since $\mathrm{type}(\{\text{"a"}, \text{"b"}, \text{"c"}\}) = (\mathsf{Set}, \mathsf{String})$, it implies that $\mathrm{type}(\{\text{"a"}, \text{"b"}, \text{"c"}\}) \preccurlyeq (\mathsf{Set}, \mathsf{State})$;
- Since $|\ \{\text{"a"}, \text{"b"}, \text{"c"}\}\ | = 3$, it implies that $1 \leqslant |\ \{\text{"a"}, \text{"b"}, \text{"c"}\}\ | \leqslant \star$;
- alphabet does not declare an opposite so the last check is irrelevant.

states We have $\sigma^{abc}(\mathsf{states}) = \langle\!\langle 1, 2, 3 \rangle\!\rangle$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{states}) = (\bot, \bot, (1, \star, \mathsf{OSet}, \mathsf{State}), \mathsf{fsm})$.

- Since $\mathrm{type}(\langle\!\langle 1, 2, 3 \rangle\!\rangle) = (\mathsf{OSet}, (\mathsf{FSM}, \mathsf{State}))$, it implies that $\mathrm{type}(\langle\!\langle 1, 2, 3 \rangle\!\rangle) \preccurlyeq (\mathsf{OSet}, \mathsf{String})$;
- Since $|\ \langle\!\langle 1, 2, 3 \rangle\!\rangle\ | = 3$, it implies that $1 \leqslant |\ \langle\!\langle 1, 2, 3 \rangle\!\rangle\ | \leqslant \star$;
- We have $\sigma^1(\mathsf{fsm}) = abc$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{states})$ and effectively, $abc \in \sigma^1(\mathsf{fsm})$ (same holds for 2's and 3's values).

transitions We have $\sigma^{abc}(\mathsf{transitions}) = \langle\!\langle a, b, c \rangle\!\rangle$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) = (\bot, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}), \mathsf{fsm})$.

- Since $\mathrm{type}(\langle\!\langle a, b, c \rangle\!\rangle) = (\mathsf{OSet}, (\mathsf{FSM}, \mathsf{State}))$, it implies that $\mathrm{type}(\langle\!\langle a, b, c \rangle\!\rangle) \preccurlyeq (\mathsf{OSet}, \mathsf{String})$;
- Since $|\ \langle\!\langle a, b, c \rangle\!\rangle\ | = 3$, it implies that $1 \leqslant |\ \langle\!\langle a, b, c \rangle\!\rangle\ | \leqslant \star$;
- We have $\sigma^a(\mathsf{fsm}) = abc$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions})$ and effectively, $abc \in \sigma^a(\mathsf{fsm})$ (same holds for a's and b's values).

State A State instance has five accessible properties: $\mathrm{Dom}\,(\pi(\mathsf{FSM})(\mathsf{State})) = \{\mathsf{label}, \mathsf{kind}, \mathsf{in}, \mathsf{out}, \mathsf{fsm}\}$. Properties in and out only differ for their opposite, therefore we only consider in.

label We have $\sigma^1(\mathsf{label}) = \text{"1"}$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$.

- Since $\mathrm{type}(\text{"1"}) = (\bot, \mathsf{String})$, it implies that $\mathrm{type}(\text{"1"}) \preccurlyeq (\bot, \mathsf{String})$;
- Since $|\ \text{"1"}\ | = 1$, it implies that $1 \leqslant |\ \text{"1"}\ | \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

kind We have $\sigma^1(\mathsf{kind}) = \mathsf{START}$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{kind}) = (\top, \bot, (1, 1, \bot, \mathsf{Kind}), \bot)$.

- Since $\mathrm{type}(\mathsf{START}) = (\bot, \mathsf{Kind})$, it implies that $\mathrm{type}(\mathsf{START}) \preccurlyeq (\bot, \mathsf{Kind})$;
- Since $|\ \mathsf{START}\ | = 1$, it implies that $1 \leqslant |\ \mathsf{START}\ | \leqslant 1$;
- kind does not declare an opposite so the last check is irrelevant.

in We have $\sigma^1(\mathsf{in}) = \langle\!\langle b \rangle\!\rangle$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) = (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt})$.

- Since $\mathrm{type}(\langle\!\langle b \rangle\!\rangle) = (\mathsf{OSet}, \mathsf{Transition})$, it implies that $\mathrm{type}(\langle\!\langle b \rangle\!\rangle) \preccurlyeq (\mathsf{OSet}, \mathsf{Transition})$;
- Since $|\ \langle\!\langle b \rangle\!\rangle\ | = 1$, it implies that $0 \leqslant |\ \langle\!\langle b \rangle\!\rangle\ | \leqslant \star$;
- We have $\sigma^b(\mathsf{tgt}) = 1$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) = (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})$ and effectively, $1 \in \sigma^b(\mathsf{tgt})$.

fsm We have $\sigma^1(\mathsf{fsm}) = abc$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{fsm}) = (\bot, \bot, (1, 1, \bot, \mathsf{FSM}), \mathsf{transitions})$.

- Since $\mathrm{type}(abc) = (\bot, (\mathsf{FSM}, \mathsf{FSM}))$, it implies that $\mathrm{type}(abc) \preccurlyeq (\bot, (\mathsf{FSM}, \mathsf{FSM}))$;
- Since $|\ abc\ | = 1$, it implies that $\leqslant |\ abc\ | \leqslant 1$;
- We already saw that $\sigma^{abc}(\mathsf{transitions}) = \langle\!\langle a, b, c \rangle\!\rangle$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{FSM})(\mathsf{transitions}) = (\bot, \bot, (0, \star, \mathsf{OSet}, \mathsf{Transition}$ and effectively, $b \in \sigma^{abc}(\mathsf{transitions})$.

Transition A Transition has four accessible properties: $\mathrm{Dom}\,(\pi(\mathsf{FSM})(\mathsf{Transition})) = \{\mathsf{label}, \mathsf{src}, \mathsf{tgt}, \mathsf{fsm}\}$. Properties src and tgt only differ for their opposite, therefore we only consider tgt (as the opposite of property in for type State).

label We have $\sigma^b(\mathsf{label}) = \text{"b"}$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Label})(\mathsf{label}) = (\top, \bot, (1, 1, \bot, \mathsf{String}), \bot)$.

- Since $\mathrm{type}(\text{"b"}) = (\bot, \mathsf{String})$, it implies that $\mathrm{type}(\text{"b"}) \preccurlyeq (\bot, \mathsf{String})$;
- Since $|\ \text{"b"}\ | = 1$, it implies that $1 \leqslant |\ \text{"b"}\ | \leqslant 1$;
- label does not declare an opposite so the last check is irrelevant.

tgt We have $\sigma^b(\mathsf{tgt}) = 1$ and $\mathrm{prop}(\mathsf{FSM})(\mathsf{Transition})(\mathsf{tgt}) = (\bot, \bot, (1, 1, \bot, \mathsf{State}), \mathsf{in})$.

- Since $\mathrm{type}(1) = (\bot, (\mathsf{FSM}, \mathsf{State}))$, it implies that $\mathrm{type}(1) \preccurlyeq (\bot, (\mathsf{FSM}, \mathsf{State}))$;
- Since $|\ 1\ | = 1$, it implies that $\leqslant |\ 1\ | \leqslant 1$;
- We already saw that $\sigma^1(\mathsf{in}) = \langle\!\langle b \rangle\!\rangle$ with $\mathrm{prop}(\mathsf{FSM})(\mathsf{State})(\mathsf{in}) = (\bot, \bot, (0, \star, \mathsf{List}, \mathsf{Transition}), \mathsf{tgt})$ and effectively, $b \in \sigma^1(\mathsf{in})$.

3.7 Discussions

This Section first discusses some of the semantics' design choices and Kermeta's specific constructions, and then compares with related works in the domain of the Structural Language.

3.7.1 Design Choices

Names. The proposes formalisation heavily relies on names: the sets $\mathcal{M}$ of metamodels and $\mathbb{M}$ of models bind names with respectively types and values. A first point of discussion is the way names are extracted from metamodels. We briefly discussed in Sec. 3.2.1 and illustrated the extraction process from the graphical representation on our running example in Sec. 3.6. The extraction process from the textual representation is further simplified by the classical use of namespaces into BNF grammars. Moreover, we introduced two simplifications on package and enumeration literal names: each of these names is always prefixed by another name that is guaranteed to be unique (the "path" of package names, and the enumeration name, respectively). These simplifications seem reasonable since they are used in Kermeta itself (cf. Appendix A). A last point is the use of property namespaces in Kermeta: the keyword **attribute** introduce a property (i.e. either a MOF attribute or a reference) that is contained. We reflected this point in our constraint on the definition of $\mathcal{P}\mathrm{rop}$.

The choice of using names to build the definitions of $\mathcal{M}$ and $\mathbb{M}$ can easily be criticised in the sole context of a structural part's formalisation: it is easier to consider feature *identifiers*, reflecting what is behind MOF. Identifiers also avoid using qualified names, building the functions π_{MM} and ω_{MM}, and using the disambiguation clause **from**. As a counterpart, they introduce an unnecessary burden when dealing with the Action Language, which relies explicitly on feature names and consequently either forces to change the Action Language itself to take advantage of identifiers, or requires an explicit mapping from identifiers to names, which is exactly the reason why identifiers are usually used.

Types. The types formalised in the set MType do not trustfully represent Kermeta's types. First, primitive types are not "*primitive*" but are rather meta-represented as classes in the package `kermeta::standard`: they all subclass `ValueType`, with eventually umbrella classes to wrap common features (as `Numeric` for numerical types Integer and Real). As a concrete consequence, they do not respect the expected Hasse Diagram of Def. 3.2 and require a proper API to implement that. Furthermore, some of these types do not possess a "surface" syntax and should be obtain by using API features (e.g., var x: Real init "1.0".toReal to obtain the real value 1.0).

Similarly, collections representation in Kermeta does not follow the natural Hasse Diagram of Def. 3.2 but rather adopts a hierarchy that favours implementation and code generation purposes (cf. [12, §2.14]): for example, the abstract class `OrderedCollection`, from which `OrderedSet` and `Sequence` inherit, factorises common features like iterators. These choices reflect more implementation efficency choices than theoretical implications.

3.7.2 Related Works

We discuss in this Section the related works that focus on structural parts of a DSML; those that directly relate to DSML transformations and DSML behavior are postponed to Chapter 4.

Historically, most formalisations focused on structural aspects, which is an understandable and natural first step. Several formal frameworks were used, but accordingly to our initial motivation, they contrast with our work by the fact they are expressed in the syntax of a particular tool. Moreover, some of them are incomplete regarding MOF.

As already mentioned, Algebraic Specifications (AS) were used to formalise MOF in [5] together with its reflection capabilities, using MAUDE. Abstract State Machines (ASM) serve both as formal foundations for MOF [15] and as a metamodeling framework for ASM [16], thus providing bridges between MDE and ASM tools. The Z language was used in [30] but does not cover packages, and stay very general for attributes/references and do not take into consideration collection/multiplicity types, despite the fact that Z offers these concepts natively.

Category Theory is another important framework: Constructive Type Theory was used in [25] to formalise MOF in the context of the "proofs-as-programs" concept, but without an explicit notion of containment/opposite references, and do not provide a clear mechanism to transform MOF specification into Constructive Type Theory; Graph-Based (GB) formalisations are used to achieve both formalisation of MOF and transformations [29, 4], but usual MOF constructions like containment and inheritance were not addressed until recently [18].

Numerous contributions try to formalise UML diagrams (and in particular, as expected, UML Class Diagrams) by translating them into languages with well-defined semantics. As a general remark, these contributions always address specific parts or small subsets of UML Diagrams. Among many other formalisms, we already mentioned Z [30], but also Object-Z in [31], the algebraic language CASL [27] where the behavior is expressed with transition systems that naturally flow from algebraic specifications, and the theorem-prover PVS by abstracting Diagrams into state predicates in [20].

Chapter 4

Action Language

Kermeta's AL is structurally weaved into the SL by extending the class *Operation* with a *Body* (cf. Fig. 2.1), which consists in a sequence of statements (cf. [21, 12]). This Chapter starts by reminding the restrictions we consider on the original Kermeta AL. Following these restrictions, the Chapter proceeds by defining a *core* AL for Kermeta, sufficient to represent sufficiently rich Kermeta's actions to handle a large variety of transformations: this core AL can be seen as an intermediate representation for Kermeta operations' bodies if the actions used comply with our subset AL. We will refer to our subset AL as *core* AL whereas Kermeta's AL will be refered to as *original*.

The Chapter then proceeds by formally specifying the AL's semantics using the classical SOS framework [35]: from the definition of the AL abstract syntax, we present a type-checking system in Sec. 4.3 that ensures the correct use of the statements; and finally provide the rewriting rules in Sec. 4.4.

4.1 Restrictions

Figure 4.1 depicts (an excerpt of) Kermeta's original AL, as presented in the Manual [12, §3.3] (where the AL is called "*Behavior Package*" and has one superclass named *Expression* from which all other constructions inherit).

We do not address *genericity* and *model type* because it heavily complicates the concise definition of the semantic domain: the classes *TypeLiteral* (for literal representing types), *LambdaParameter* (representing parameters that uses a generic type), *TypeReference* (for referencing a type) and *LambdaExpression* (for defining expressions that uses generics) are not part of our AL.

Since exception handling mechanisms concerns illed behaviors of transformations, and because we target formal verification at the hand, we do not consider exception constructions: the classes *Raise* (for raising an exception inside an operation's body) and *Rescue* (for declaring an exceptions attached to an operation) do not appear in our AL.

Finally, we do not consider *native calls* to Java (cf. *JavaStaticCall*) as it concerns more interaction with the platform.

Two other classes of this metamodel do not appear formally in our metamodel, namely *Block* and *VariableDecl*, because they are represented otherwise (see details below). All other classes from the *Expression* behavioral metamodel of Kermeta are handled in our core AL.

4.2 Definition

In Kermeta, the operation's body consists in an *ordered* sequence of *unique statements* (in the sense that, despite the possibility to have structurally identical statements, the statements themselves are different entities). Kermeta's statements consists in either *local variable declarations* or *statements* with an actual dynamic effect. We first explain how local variables are represented before proceeding to the statements' syntax.

4.2.1 Local Variable Declarations

This formal semantics focuses on the dynamic aspects of Kermeta's AL. We operate several syntactic simplifications that do not change the dynamic semantics but allows us to simplify the presentation without any loss of generality.

The first simplification concerns the variable declarations in an operation's body, which are all shifted at the beginning before any other statement (variables can always be renamed consistently to avoid name clashes), and they are considered

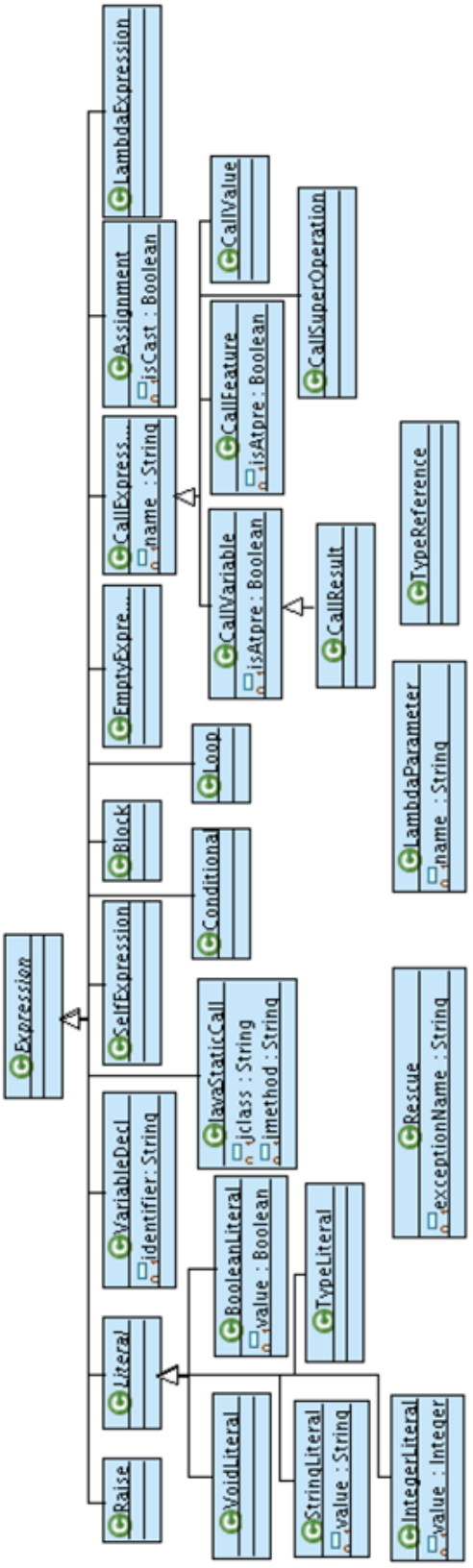


Figure 4.1: Kermeta's Action Language (from [12, §3.3])

			Body	::=	[Stm]$^+$
			Stm	::=	*lab:* Stmt
Exp	::=	**null** | scalarExp | instExp | collExp	Stmt	::=	condStmt
ScalarExp	::=	literal | instExp **instanceof** pClassN		|	assignStmt | castStmt
InstExp	::=	**self** | lhs		|	newInstStmt | CollStmt
Lhs	::=	varN | paramN		|	returnStmt | CallStmt
	|	target.propN	CondStmt	::=	**if** exp
Target	|	**super** | instExp	AssignStmt	::=	lhs **=** exp
CollExp	|	exp.nativeExp	CastStmt	::=	varN **?=** exp
NativeExp	|	*isEmpty()* | *size()* | *at(*exp*)*	NewInstStmt	::=	varN **=** pClassN***.new***
			ReturnStmt	::=	**return** | **return** exp
LocalN	::=	varN | paramN | **self**	CallStmt	::=	call | varN **=** call
PClassN	::=	(pkgN classN)	Call	::=	target **.**opN**(**exp*****)
			CollStmt	::=	exp.nativeStmt
			NativeStmt	::=	*add*(exp) | *del*(exp)

Figure 4.2: Expressions (left) and Statements (right)

to scope to the entire body. This classical simplification does not affect the operation's behavior but only influences the complexity of the type-checking system [1].

Definition 4.1 (Local Type Environment)**.** *Given a metamodel* $\mathsf{MM} \in \mathcal{M}$*, the* local (type) environment *is a function that associates to each operation in* MM *a function mapping local declarations to their types.*

$$\lambda_{\mathsf{MM}} : \mathsf{QOpN} \nrightarrow \mathsf{LocalN} \nrightarrow \mathsf{MType}$$

As usual, the attached metamodel MM will be omitted, since statements are always structurally attached to an operation of a class in MM. Notice also that if a operation is abstract, no local variables are defined:

$$\forall \mathsf{qop} \in \mathsf{QOpN}, \mathrm{abs}_{\mathsf{MM}}(\mathsf{qop}) \implies \mathsf{Dom}\left(\lambda_{\mathsf{MM}}(\mathsf{qop})\right) = \varnothing$$

4.2.2 Syntax

Starting from Kermeta's AL and removing the constructions reminded in Sec. 4.1, we obtain the core AL presented in Fig. 4.2 in the form of a BNF grammar. Using a grammar was a choice made for two main reasons: first, traditional SOS rules are usually expressed on top of BNF sentences; and second, the definition of the core AL is more compact in this form (as compared to the original metamodel in the Manual). The non-terminal Body of Fig. 4.2 is directly linked with the metamodel's class *Body* in Fig. 2.1, which formally establishes the connection between the SL and the AL. Furthermore, the grammar presented here follows as much as possible the textual concrete syntax of Kermeta, which make it easier to retrieve the corresponding constructions.

The Body consists in a non-empty sequence of statements Stm: we already discarded from operations functions the case where an operation's body is empty (cf. definition in Sec. 3.3.5). Each statement is associated with a label $\mathsf{lab} \in \mathsf{Lab}$ that uniquely identify the statement through the entire metamodel. Consequently, we can consider that an operation's body is represented by the sequence of its labels, given by the function $\mathrm{labs}_{\mathsf{MM}}$. A *label encloser* function *op* associates a label to its enclosing operation in the metamodel[1].

$$\begin{array}{rcl} \mathrm{labs}_{\mathsf{MM}} & : & \mathsf{QOpN} \longrightarrow \langle\!\langle \mathsf{Lab} \rangle\!\rangle \\ op_{\mathsf{MM}} & : & \mathsf{Lab} \nrightarrow \mathsf{QOpN} \end{array}$$

The core AL is divided in two syntactic groups: expressions Exp are side-effect free and simply evaluate to a value; they are used inside statements Stmt that actually alter the model under execution. This simplification is not new [1, 32, 26] and is convenient to clarify the semantics of OO languages.

Expressions are of four kinds: the **null** expression; *scalar* expressions ScalarExp; *instance* expressions InstExp; or *collection* expressions.

[1] Consider a metamodel $\mathsf{MM} = (p, c, e, \mathrm{prop}, o) \in \mathcal{M}$ where for example $o(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) = (\_, \_, \_, \_, \mathsf{Body})$. From the definition in Fig. 4.1, we consider that $\mathsf{Body} = \langle\!\langle \mathsf{lab}_1, \ldots, \mathsf{lab}_n \rangle\!\rangle$. Then labs and op are reverse functions, i.e. $\mathrm{labs}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) = \mathsf{Body} \iff \forall i, \mathrm{op}(\mathsf{lab}_i) = (\mathsf{pkg}, \mathsf{c}, \mathsf{op})$.

Lab	While		Flattened	If-then-else	
	C.F	**Original**	**Representation**	**Original**	**C.F**
0	$(1, \mathsf{n}+1)$	while(...){	if(...){	if(...){	$(1, \mathsf{n}+1)$
1	$(2, \perp)$	stm_1	stm_1	stm_1	$(2, \perp)$
	...	...	...	...	...
n	$(0, \perp)$	stm_n	stm_n	stm_n	$(\mathsf{n}+\mathsf{m}+1, \perp)$
		}		}else{	
n+1	$(\mathsf{n}+2, \perp)$	stm_{n+1}	stm_{n+1}	stm_{n+1}	$(\mathsf{n}+2, \perp)$
					...
n+m	$(\mathsf{n}+\mathsf{m}+1, \perp)$	stm_{n+m}	stm_{n+m}	stm_{n+m}	$(\mathsf{n}+\mathsf{m}+1, \perp)$
				}	
n+m+1		stm_{n+m+1}	stm_{n+m+1}		stm_{n+m+1}

Table 4.1: Common representation of traditional `while`/`if-then-else` statements with CondStmt.

A scalar expression is either a literal Literal or an **instanceof** expression. The terminal Literal abstracts from the actual representation of literal and associated operations. An **instanceof** expression consists of an instance expression and a declared class in the form of a package name and a class name, thus reflecting in the grammar the definition of PClassN.

Instance expressions have values that designate "assignable" entities: either **self**, or a left-hand side expression. Left-hand side expressions Lhs are either a variable / parameter access, or a property access at the level of the superclass or by navigating through an instance.

Collection expressions consists in an instance expression that designates a collection, and a native expression, which is one of the following: *size*() that returns the size of a collection; *isEmpty*() that returns true iff a collection is empty; or *at*(exp) that returns an element in an ordered collection at the position given by the expression.

An extra syntactic set LocalN is defined, which is not strictly part of the grammar definition, but usefully collects *local names* inside an operation's body for easing the semantics' formalisation.

Statements are of five kinds: *conditional*, *cast* and *assignment*, *instance creation*, *operation call* management, and *collection* statements. A *conditional* CondStmt is reduced to a conditional branching statement. An *assignment* AssignStmt assigns an expression to a left-hand side expression. A *cast* CastStmt casts an expression to a variable. An *instance creation* NewInstStmt assigns to a variable a fresh instance of a specified class, again through the form of a PClassN. A *return* ReturnStmt is either the simple keyword **return**, or this keyword used with a return expression. An *operation call* CallStmt is either a simple call, or it can return a value which is assigned to a variable. A *collection* statement CollStmt consists in a call expression (before the .) that designates a collection, on which one of the following native statement is applied: *add*(exp) or *del*(exp) respectively adds or removes an element given by the parameter expression (between parenthesis).

4.2.3 Control Flow

Notice first that our core AL does not possess any structuring statement, but rather uses only one conditional branching statement CondStmt. This simplification allows us to avoid well-studied issues that traditionally occur when using imperative constructions like iterative (`while`, `do`, for) or conditional (`if-then-else`) statements [1, 32, 26], such as variable scopes and nested control flows. Instead, statements contained in such constructions are flattened.

Definition 4.2 (Control Flow)**.** *The control flow function* $\mathrm{nxt}_{\mathsf{MM}}$ *attached to a metamodel* MM *is a function that associates to each label of the metamodel the next labels to proceed.*

$$\mathrm{nxt}_{\mathsf{MM}} : \mathsf{Lab} \longrightarrow \mathsf{Lab} \times \mathsf{Lab}_{\perp}$$

Consider a labeled statement lab : stmt $\in$ Stm. Most statements have exactly one following statement, the following one in the declaration order: in such case, $nxt(\mathsf{lab}) = (\mathsf{lab}, \perp)$. If stmt $\in$ ReturnStmt is a return statement, the control flow escapes from the operation and no following statement is expected: $nxt(\mathsf{lab}) = (\perp, \perp)$. If stmt $\in$ CondStmt is a conditional statement, there is two possible following statements, depending on the value of the conditional: $nxt(\mathsf{lab}) = (\mathsf{lab}', \mathsf{lab}'')$, where lab' (resp. lab") is the statement to which to proceed if evaluated to true (resp., false).

Table 4.1 shows how our single branching statement CondStmt acts as a normal form for the classical `while` and `if-then-else`: they have a common representation but differ only in the definition of their associated control flow

```
var trans: oset Transition [0..*]
var current: Transition
var i: Integer
01: trans := self.out
02: current := trans.at(0)
03: i := 0
04: if(trans == null) {
05:     return null
    }else{
06:     while(i == trans.size() or
          trans.at(i).label == letter){
07:         i := i+1
        }
08:     if(current == null) {
09:         return null
        }else{
10:         return current.tgt
        }
    }
```

```
var trans: oset Transition [0..*]
var current: Transition
var i: Integer
01: trans := self.out
02: current := trans.at(0)
03: i := 0
04: if(trans == null)
05: return null

06: if(i == trans.size() or
        trans.at(i).label == letter)
07: i := i+1

08: if(current == null)
09: return null

10: return current.tgt
```

$\mathsf{L}_{\mathsf{fire}}$	nxt
01	$(02, \bot)$
02	$(03, \bot)$
03	$(04, \bot)$
04	$(05, 06)$
05	$(\bot, \bot)$
06	$(07, 08)$
07	$(06, \bot)$
08	$(09, 10)$
09	$(\bot, \bot)$
10	$(\bot, \bot)$

Figure 4.3: The *fire* operation's body: at the left, declarations and associated initialisations are shifted at the beginning, and loops and *result* statements are transformed; in the middle, statements are flattened, according to Tab. 4.1; at the right, the control flow is explicited.

function nxt (columns **CF** showing $\mathrm{nxt}(\mathsf{lab})$, where lab is given in first column). Other iterative statements like `do` or `for` can syntactically be reduced to a `while` (cf. [1]).

4.2.4 Example

Figure 4.3 shows how the body of the *fire* operation is transformed into our core AL (the *accept* operation uses the same statements, namely conditionals, loops and assignments). First, all declarations are shifted (and renamed if necessary) at the beginning of the body. Second, Kermeta's loops are translated into `while` loops; and **result** assignments are replaced by corresponding **return** statements. Third, loops are flattened according to Table 4.1; and statements are labeled and the corresponding control flow is computed.

Let $\mathsf{L}_{\mathsf{fire}} \subseteq \mathsf{Lab}$ be the set of labels used in the *fire* operation. The *fire* local environment's domain is $\mathsf{Dom}\,(\lambda_{\mathsf{FSM}}(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})) = \{\mathsf{trans}, \mathsf{current}, \mathsf{i}, \mathbf{self}, \mathsf{letter}\}$. If we remember that $o_{\mathsf{FSM}}(\mathsf{FSM})(\mathsf{State})(\mathsf{fire}) = (\bot, \bot, \langle\!\langle(\mathsf{letter}, (0, 1, \bot, \mathsf{String}))\rangle\!\rangle, (0, 1, \bot, \mathsf{State}), b_{\mathsf{fire}})$, then we have the following definitions:

$$
\begin{aligned}
\forall \mathsf{lab} \in \mathsf{L}_{\mathsf{fire}}, op(\mathsf{lab}) &= (\mathsf{FSM}, \mathsf{State}, \mathsf{fire}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{trans}) &= (0, \star, \mathsf{OSet}, \mathsf{Transition}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{current}) &= (1, 1, \bot, \mathsf{Transition}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{i}) &= (1, 1, \bot, \mathsf{Integer}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathsf{letter}) &= (0, 1, \bot, \mathsf{String}) \\
\lambda(\mathsf{FSM}, \mathsf{State}, \mathsf{fire})(\mathbf{self}) &= (1, 1, \bot, \mathsf{State})
\end{aligned}
$$

4.3 Type-Checking System

A Type-Checking system [7] defines rules ensuring that statements in an operation's body are consistant with the meta-model's $\mathsf{MM} \in \mathcal{M}$ and local λ_{MM} declarations.

In order to have a sound type system, the definition of Type has to be extended to take care of the **null** value, which was not part of the Structural Language. From now on, the following definitions replace the ones given in Def. 3.2:

Definition 4.3 (Extended Types)**.** *The set of (basic) types* Type *is either a primitive type, or a class name or an enumeration name declared in the scope of a package, or the special type* $\mathbf{Null}$*, wich contains only one value,* $\mathbf{null}$*. The extended order* $\leqslant_{\mathsf{Type}}$ *is redefined for making* $\mathbf{Null}$ *specialising any other type.*

$$\mathsf{Type} \triangleq \mathsf{PClassN} \cup \mathsf{DataType} \cup \{\mathbf{Null}\}$$
$$\forall \mathsf{t}, \mathsf{t}' \in \mathsf{Type}, \mathsf{t} \preccurlyeq_{\mathsf{Type}} \mathsf{t}' \iff \mathsf{t} = \mathbf{Null} \vee \mathsf{t} \preccurlyeq_{\mathsf{Class}} \mathsf{t}' \vee \mathsf{t} \preccurlyeq_{\mathsf{Prim}} \mathsf{t}'$$

4.3.1 Expressions

A judgement "$\lambda \bullet \mathsf{qop} \vdash^{\mathsf{MM}}_{\mathsf{Expr}} \mathsf{exp} \rhd \mathsf{t}$" means that in MM, the expression $\mathsf{exp} \in \mathsf{Exp}$ appears in the body of the operation $\mathsf{qop} = (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \in \mathsf{QOpN}$, and is of type $\mathsf{t} \in \mathsf{CType}$.

The judgement $\vdash_{\mathsf{Expr}}$ is defined by structural induction. The following rules just transfer into specific rules for each kind of expressions: scalar expressions should be typed by a DataType; instance expressions should be typed by a class PClassN. The **null** expression has obviously the type **Null**.

Exp ::= **null** | scalarExp | instExp | collExp

$$\frac{}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathbf{null} \rhd (\bot, \mathbf{Null})}$$

$$\frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Scalar}} \mathsf{scalarExp} \rhd (\bot, \mathsf{t}) \quad \mathsf{t} \in \mathsf{DataType}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{scalarExp} \rhd (\bot, \mathsf{t})} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathsf{instExp} \rhd \mathsf{ct} \in \mathsf{CType}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{instExp} \rhd \mathsf{ct}} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Coll}} \mathsf{collExp} \rhd \mathsf{ct} \in \mathsf{CType}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{collExp} \rhd \mathsf{ct}}$$

Scalar Expressions. The type of a literal expression is straightforward. The type of an **instanceof** expression is Boolean if the type of its instance expression is a class type that specialises the declared class.

ScalarExp ::= literal | instanceExp **instanceof** className

$$\frac{}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Scalar}} \mathsf{lit}_{\mathsf{c},\mathsf{t}} \rhd (\mathsf{c}, \mathsf{t})} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathsf{instExp} \rhd \mathsf{c}) \quad \mathsf{c} \preccurlyeq (\mathsf{pkgN}, \mathsf{classN})}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Scalar}} \mathsf{instExp}\ \mathbf{instanceof}\ (\mathsf{pkgN}, \mathsf{classN}) \rhd (\bot, \mathsf{Boolean})}$$

Instance Expressions. The type of **self** is the declared type in the local environment λ_{MM} in op's scope. The type of a left-hand side expression is computed by a specific judgment $\vdash_{\mathsf{Lhs}}$.

instExp ::= **self**| lhs

$$\frac{(\_, \mathsf{ct}) = \lambda(\mathsf{qop})(\mathbf{self})}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathbf{self} \rhd \mathsf{ct}} \qquad \frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Lhs}} \mathsf{lhs} \rhd \mathsf{ct}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Inst}} \mathsf{lhs} \rhd \mathsf{ct}}$$

The type of a variable/parameter access is also the declared type in λ_{MM} in op's scope.

Lhs ::= varN | paramN | ...

$$\frac{\mathsf{name} \in \mathsf{VarN} \cup \mathsf{ParamN} \quad (\_, \_, \mathsf{ct}) = \lambda(\mathsf{qop})(\mathsf{name})}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Lhs}} \mathsf{name} \rhd \mathsf{ct}}$$

The type of a property access depends on the target expression. First, the class c' corresponding to the target is computed. Then, it depends of the fact that propN is ambiguous or not in the context of c' (given by from value). If not, then there must be a class c" in the inheritance hierarchy of c that has propN in its scope, i.e either declares it or inherits it (which fact is encapsulated in π). If it is ambiguous, then there must exist a disambiguation class c" from which the property declaration is retrieved (also through π).

Lhs ::= ... | **super.**propN | instExp.propN

$$\frac{\begin{array}{rcl}\mathrm{from}(\mathsf{pkg})(\mathsf{c})(\mathsf{propN}) & = & \bot \\ \mathsf{c}' & \in & \mathrm{super}(\mathsf{pkg},\mathsf{c}) \\ \mathsf{propN} & \in & \mathrm{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}')) \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN})\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \textbf{super.}\mathsf{propN} \rhd \mathsf{ct}}$$

$$\frac{\begin{array}{rcl}\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Inst}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) & = & \bot \\ \mathsf{c}'' & \in & \mathrm{super}(\mathsf{pkg},\mathsf{c}) \\ \mathsf{propN} & \in & \mathrm{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c}'')) \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}'')(\mathsf{propN})\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \mathsf{instExp}\ \textbf{.}\ \mathsf{propN} \rhd \mathsf{ct}}$$

$$\frac{\begin{array}{rcl}\mathrm{from}(\mathsf{pkg})(\mathsf{c})(\mathsf{propN}) & = & \mathsf{c}' \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN})\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \textbf{super.}\mathsf{propN} \rhd \mathsf{ct}}$$

$$\frac{\begin{array}{rcl}\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) & \vdash_{\mathsf{Inst}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg})(\mathsf{c}')(\mathsf{propN}) & = & \mathsf{c}'' \\ (\_,\_,(\_,\mathsf{ct}),\_) & = & \pi(\mathsf{pkg})(\mathsf{c}'')(\mathsf{propN})\end{array}}{\lambda \bullet (\mathsf{pkg},\mathsf{c},\mathsf{op}) \vdash_{\mathsf{Lhs}} \mathsf{instExp}\ \textbf{.}\ \mathsf{propN} \rhd \mathsf{ct}}$$

Collection Expressions. Let instExp.nativeExp ∈ CollExp. First, instExp must have a collection type with an actual collection. Then, the type of *isEmpty()* is Boolean, the type of *size()* is Integer. The type of *at* is instExp basic type if the associated expression's type is Integer and instExp collection kind specialises List (i.e. must be ordered to enable indexed access).

CollExp ::= exp.*isEmpty()* | exp.size() | exp.*at(*exp*)*

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd (\mathsf{C},\_) \\ \mathsf{C} & \neq & \bot\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{exp}.\mathit{isEmpty()} \rhd (\bot, \mathsf{Boolean})}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd (\mathsf{C},\_) \\ \mathsf{C} & \neq & \bot\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{exp}.\mathit{size()} \rhd (\bot, \mathsf{Integer})}$$

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp}' \rhd (\bot, \mathsf{Integer}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd (\mathsf{C},\mathsf{t}) \\ \mathsf{C} & \neq & \bot \\ \mathsf{C} & \preccurlyeq & \mathsf{List}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{exp}.\mathit{at(}\mathsf{exp}'\mathit{)} \rhd (\bot, \mathsf{t})}$$

4.3.2 Statements

A judgement "$\lambda \bullet \mathsf{qop} \vdash^{\mathsf{MM}}_{\mathsf{Stm}} \mathsf{stm}$" means that the statement stm ∈ Stm appears inside the body of the operation qop ∈ QOpN, and is well-formed.

A statement stm is well-formed if its inner statement stmt is well-formed and its labels are consistent, i.e. stm's execution proceeds to statements within the same operation's body.

Stm ::= *lab:* stmt

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Stmt}} & \mathsf{stmt} \\ \mathrm{nxt}(\mathsf{lab}) & = & (\mathsf{lab}',\mathsf{lab}'') \\ \mathit{op}(\mathsf{lab}') & = & \mathsf{qop} \\ \mathit{op}(\mathsf{lab}'') & = & \mathsf{qop}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{lab} : \mathsf{stmt}}$$

The previous rule uses a judgement $\vdash_{\mathsf{Stmt}}$ defined by structural induction. A conditional statement is well-formed if its expression's type is Boolean.

CondStmt ::= if exp

$$\frac{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{exp} \rhd (\bot, \mathsf{Boolean})}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{if}\ \mathsf{exp}}$$

An instance creation statement is well-formed if the created instance's type specialises the class type declaration.

NewInstStmt ::= var **= new** PClassN

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{var} \rhd (\bot, \mathsf{c}) \\ \mathsf{c} & \in & \mathsf{PClassN} \\ \mathsf{c} & \preccurlyeq & (\mathsf{pkgN},\mathsf{classN})\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{var} = (\mathsf{pkgN}\ \mathsf{classN})\textbf{.}\textit{\textbf{new}}}$$

A return statement with (resp. without) an expression is well-formed if it appears inside the body of an operation whose return type specialises its expression's type (resp. is void).

ReturnStmt ::= **return** | **return** exp

$$\frac{\mathrm{type}(\mathsf{qop}) = (\bot, \bot, \bot)}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{return}}$$

$$\frac{\begin{array}{rcl}\mathrm{type}(\mathsf{qop}) & = & (\bot, \bot, \mathsf{t}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \textbf{return}\ \mathsf{exp}}$$

Assignment and Casting statements are similar. An assigment statement is well-typed if the type of the right-hand side expression specialises the type of the left-hand side expression. A cast statement is well-typed if the types of the left-hand side and the right-hand side are comparable (i.e. they have a "common supertype" [12, §2.9.2.1]).

AssignStmt ::= lhs **=** exp

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{lhs} \rhd \mathsf{t} \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{lhs} = \mathsf{exp}}$$

CastStmt ::= varN **?=** exp

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{varN} \rhd \mathsf{t} \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd \mathsf{t}' \\ \mathsf{t}' \preccurlyeq \mathsf{t} & \vee & \mathsf{t} \preccurlyeq \mathsf{t}'\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{varN}\ \textbf{?=}\ \mathsf{exp}}$$

An Operation Call statement is typed in two steps. The first step assumes another judgement $\vdash_{\mathsf{Call}}$ explained hereafter. A call without assignment is well-typed if the call expression is well-typed and the operation has no return type; and a call with assignment is well-typed if the call expression is well-typed and the return type of the operation specialises the type of the assigned variable.

CallStmt ::= target.opN(exp*)

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Call}} & \mathsf{target.opN}(\mathsf{exp}^*) \\ \mathrm{type}(\mathsf{qop}) & = & \bot\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{target.opN}(\mathsf{exp}^*)}$$

CallStmt ::= varN = target.opN(exp*)

$$\frac{\begin{array}{rcl}\lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Call}} & \mathsf{target.opN}(\mathsf{exp}^*) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{varN} \rhd \mathsf{t} \\ \mathrm{type}(\mathsf{qop}) & \preccurlyeq & \mathsf{t}\end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{varN} = \mathsf{target.opN}(\mathsf{exp}^*)}$$

The following set of rules deals with the call itself (for the judgement $\vdash_{\mathsf{Call}}$). First, the type of the target expression should denote a class (which is obviously the case for **super**). Second, all expressions for effective parameters are also well-typed. Third, the type-checking follows the same schema as for property access because it depends on the target and the possible multiply inherited method name: if the method name is ambiguous, then the method call is well-typed if there exists a method with the same name in the inheritance hierarchy of the disambiguated class; if it is not, then the method call is well-typed if there exists one class in the inheritance hierarchy that declares a method with the same name. The class from which the method name is looked for depends on the target: if the target is **super**, then the method name is looked for from the superclass; otherwise, it is looked from the class to which the instance target is typed. Finally, the types of the effective parameters expressions must specialise the formal parameters' types (in order).

Call ::= instExp.opN(exp*)

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i} \\ \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \bot \\ \mathsf{opN} & \in & \mathrm{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}')) \\ \mathrm{partypes}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \langle\!\langle (\_, \_, \mathsf{ct}'_1), \ldots, (\_, \_, \mathsf{ct}'_n) \rangle\!\rangle \\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \vdash_{\mathsf{Call}} \mathsf{instExp.opN}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)}$$

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i} \\ \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{instExp} \rhd \mathsf{c}' \\ \mathrm{from}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \mathsf{c}'' \\ \mathsf{opN} & \in & \mathrm{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}'')) \\ \mathrm{partypes}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \langle\!\langle (\_, \_, \mathsf{ct}'_1), \ldots, (\_, \_, \mathsf{ct}'_n) \rangle\!\rangle \\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg}, \mathsf{class}, \mathsf{op}) \vdash_{\mathsf{Call}} \mathsf{instExp.opN}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)}$$

Call ::= **super.**opN(exp*)

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{ct_i} \\ \mathrm{from}(\mathsf{pkg}, \mathsf{c}, \mathsf{opN}) & = & \bot \\ \mathsf{c}' & \in & \mathrm{super}(\mathsf{pkg}, \mathsf{c}) \\ \mathsf{opN} & \in & \mathrm{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}')) \\ \mathrm{partypes}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \langle\!\langle (\_, \_, \mathsf{ct}'_1), \ldots, (\_, \_, \mathsf{ct}'_n) \rangle\!\rangle \\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \vdash_{\mathsf{Call}} \mathbf{super}.\mathsf{opN}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)}$$

$$\frac{\begin{array}{rcl}\forall \mathsf{i}, \lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) & \vdash_{\mathsf{Exp}} & \mathsf{exp_i} \rhd \mathsf{t_i} \\ \mathrm{from}(\mathsf{pkg}, \mathsf{c}, \mathsf{opN}) & = & \mathsf{c}' \\ \mathsf{opN} & \in & \mathrm{Dom}\,(\omega(\mathsf{pkg})(\mathsf{c}')) \\ \mathrm{partypes}(\mathsf{pkg}, \mathsf{c}', \mathsf{opN}) & = & \langle\!\langle (\_, \_, \mathsf{ct}'_1), \ldots, (\_, \_, \mathsf{ct}'_n) \rangle\!\rangle \\ \forall i,\ \mathsf{ct_i} & \preccurlyeq & \mathsf{ct}'_i\end{array}}{\lambda \bullet (\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \vdash_{\mathsf{Call}} \mathbf{super}.\mathsf{opN}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)}$$

$$\begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Coll}} : \mathsf{Collection} & \longrightarrow & \mathbb{V} \\ \mathsf{Bag} & \mapsto & [] \\ \mathsf{Set} & \mapsto & \varnothing \\ \mathsf{List} & \mapsto & \langle\,\rangle \\ \mathsf{OSet} & \mapsto & \langle\!\langle\,\rangle\!\rangle \end{array} \qquad \begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Prim}} : \mathsf{PrimType} & \longrightarrow & \mathbb{V} \\ \mathsf{Boolean} & \mapsto & \bot \\ \mathsf{Integer} & \mapsto & 0 \\ \mathsf{Real} & \mapsto & 0.0 \\ \mathsf{String} & \mapsto & \text{“”} \end{array}$$

$$\begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Enum}} : \mathsf{PEnumN} & \longrightarrow & \mathbb{V} \\ (\mathsf{pkg}, \mathsf{enum}) & \mapsto & \mathrm{fst}(e(\mathsf{pkg})(\mathsf{enum})) \end{array} \qquad \begin{array}{rcl} \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Class}} : \mathsf{PClassN} & \longrightarrow & \mathbb{V} \\ (\mathsf{pkg}, \mathsf{enum}) & \mapsto & \mathbf{null} \end{array}$$

Figure 4.4: Functions computing default values for all collection types

The two collection statements are typed the same way. They are well-formed if their application expression's type has an actual collection kind, and if their parameter expression's basic type specialises the application expression's one.

$$\mathsf{CollStmt} ::= \mathsf{exp.add(exp')} \mid \mathsf{exp.del(exp')}$$

$$\frac{\begin{array}{rcl} \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp} \rhd (\mathsf{C}, \mathsf{t}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{exp}' \rhd (\bot, \mathsf{t}') \\ \mathsf{C} & \neq & \bot \\ \mathsf{t}' & \preccurlyeq & \mathsf{t} \end{array}}{\lambda \bullet \mathsf{qop} \vdash_{\mathsf{Stmt}} \mathsf{exp.\_\_\_(exp')}}$$

4.4 Semantics

An SOS specifies the semantics by means of a labeled transition system describing the abstract execution of a transformation. A *semantic domain* interprets the syntactic constructions, and is further enriched with the necessary information to describe the dynamics of the execution, resulting in an *execution state*, or *configuration* that represents the state of the transition system. Then, the transitions are simply rewriting rules between configurations [35].

4.4.1 Semantic Domain and Operations

The *semantic domain* is formally a set $(\mathbb{D}, \{op_i\}_{i \in I})$ that gives the meaning of all syntactic constructions of the language, i.e. a way to unambiguously interpret what the language describes. It generally comes with operations that facilitate the manipulation of this domain's element.

Definition 4.4 (Semantic Domain — Target)**.** *The semantic domain $\mathbb{D}$ is a pair consisting of a model and a local (store) environment. A local store environment $\mathbb{L}$ is a function mapping local names to values. A* target *is either a local name, or a pair consisting of an object and a property name.*

$$\begin{array}{rcl} \mathbb{L} & \triangleq & \mathsf{LocalN} \nrightarrow \mathbb{V} \\ \mathbb{D} & \triangleq & \mathbb{M} \times \mathbb{L} \\ \mathbb{T} & \triangleq & \mathsf{LocalN} \cup (\mathbb{O} \times \mathsf{PropN}) \end{array}$$

A *target* $t \in \mathbb{T}$ designates a element in the semantic domain that carries a value: it is either a local name, stored in the local environment; or a property within an object, stored inside the model. We then use a functional notation to access transparently these elements from the semantic domain: if $d = (m, l) \in \mathbb{D}$ is a domain, we note $d(t)$ the value stored for t, i.e. $d(t) \triangleq l(t)$ if $t \in \mathsf{LocalN}$, and $d(t) \triangleq m(o, \mathsf{p})$ if $t = (o, \mathsf{p}) \in \mathbb{O} \times \mathsf{PropN}$. Similarly, we note $d[t \mapsto v]$ the update of t by v in d, i.e. $d[t \mapsto v] \triangleq l[t \mapsto v]$ if $t \in \mathsf{LocalN}$, and $d[t \mapsto v] \triangleq m(t)$ if $t \in \mathbb{O} \times \mathsf{PropN}$.

Default Values and Instances

When calling an operation (i.e. executing a CallStmt statement), it is necessary to build a new environment where local variables are associated with default values. Similarly, when creating a new instance (i.e. executing a NewInstStmt statement), it is necessary to build a "new" valid object of the correct type.

The function $\mathfrak{Default}_{\mathsf{MM}}$ associates to each collection type the corresponding default value, by following the structural definition of CType (and as usual, we omit to mention the concerned metamodel MM when clear from context):

$$\begin{array}{rcl}
\mathfrak{Default}_{\mathsf{MM}} : \mathsf{CType} & \longrightarrow & \mathbb{V} \\
(\bot, \mathsf{pt}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Prim}}(\mathsf{pt}) \text{ if } \mathsf{e} \in \mathsf{PrimType} \\
(\bot, \mathsf{e}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Enum}}(\mathsf{e}) \text{ if } \mathsf{e} \in \mathsf{PEnumN} \\
(\bot, \mathsf{c}) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Class}}(\bot, \mathsf{c}) \text{ if } \mathsf{e} \in \mathsf{PClassN} \\
(\mathsf{C}, _) & \mapsto & \mathfrak{Default}^{\mathsf{MM}}_{\mathsf{Coll}}(\mathsf{C}) \text{ if } \mathsf{C} \in \mathsf{Collection}
\end{array}$$

The definition distinguishes the cases when the collection type has a collection or not. If there is a collection, the default value is simply the corresponding "empty" collection (emty bag for Bag, empty set for Set and so on) independently from the basic type. If there is no collection, then it depends on the basic type. For *primitive types*, the default value is predefined independently from any metamodel: it is the false value for Boolean, the zero value for Integer, and so on; for *enumeration types*, the default value for is the first enumeration literal declared for this enumeration; for *class types*, the default value for a class is simply the **null** value. Figure 4.4 gives the definitions of all intermediary default functions.

The function $\mathfrak{Initialise}_{\mathsf{MM}}$ builds a fresh "initial" instance, valid for a given class type $(\mathsf{pkg}, \mathsf{c}) \in \mathsf{PClassN}$: its type is $(\mathsf{pkg}, \mathsf{c})$ and its state is a function s_0 that associates to each accessible property its default value.

$$\begin{array}{rcl}
\mathfrak{Initialise}_{\mathsf{MM}} : \mathsf{PClassN} & \longrightarrow & \mathsf{PClassN} \times \mathbb{S}\mathsf{tate} \\
(\mathsf{pkg}, \mathsf{c}) & \mapsto & ((\mathsf{pkg}, \mathsf{c}), s_0) \\
\text{where} & & \begin{cases} \mathsf{Dom}\,(s_0) & = & \mathsf{Dom}\,(\pi(\mathsf{pkg})(\mathsf{c})) \\ s_0(\mathsf{pkg}, \mathsf{c}, \mathsf{propN}) & = & \mathfrak{Default}_{\mathsf{MM}}(\pi(\mathsf{pkg}, \mathsf{c}, \mathsf{propN})) \end{cases}
\end{array}$$

Operation Call Initialisation

Calling an operation requires to build a new local environment in order to execute the called operation's body. The function $\mathfrak{Start}$ whose result is a domain that corresponds to the starting state of a given operation: it builds a "new" local store environment and keeps the old model.

$$\begin{array}{rcl}
\mathfrak{Start} : \mathsf{QOpN} \longrightarrow \mathbb{V} \times \langle \mathbb{V} \rangle \nrightarrow \mathbb{D} & \nrightarrow & \mathbb{D} \\
\mathfrak{Start}(\mathsf{pkg}, \mathsf{c}, \mathsf{op})(v_{\mathsf{this}}, \langle v_1, \ldots, v_n \rangle)(m, l) & \mapsto & (m, l')
\end{array}$$

where $\mathsf{Dom}\,(l') = \mathsf{Dom}\,(\lambda(\mathsf{pkg}, \mathsf{c}, \mathsf{op}))$ and

$$l' = \left\{ \begin{array}{rcl} \mathsf{this} & \mapsto & v_{\mathsf{this}} \\ \mathsf{p_i} & \mapsto & v_i \ \forall \mathsf{p_i} \in \mathrm{parnames}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \\ \mathsf{varN} & \mapsto & \mathfrak{Default}(\mathsf{ct}) \text{ with } \lambda((\mathsf{pkg}, \mathsf{c}, \mathsf{varN})) = (\_, \_, \mathsf{ct}) \end{array} \right\}$$

Basically, the new local store environment respects the local declarations of the operation, and maps parameters to the corresponding value (in the order of their declaration) and variables to their default value.

Expression / Target Evaluation

As most of the statements are composed of expressions, it is necessary to provide a mechanism to properly evaluate them. Recall first that an expression is either the **null** expression, a scalar expression or an instance expression. Since expressions are designed to be side-effect free, it is possible to define a function that computes the associated value of an expression, when evaluated in the context of an model.

We first assume the existence of two functions which effect is abstracted from any particular implementation that directly depends on the execution platform: the function $[\![\bullet]\!]_{\mathrm{Lit}}$ computes the value of literals and associated operators; the function $[\![\bullet]\!]_{\mathrm{Conv}}$ adequately converts primitive types between each others (e.g. $[\![\mathsf{Real}]\!]_{\mathrm{Conv}}(1)$ converts the integer value 1 into the corresponding real value 1.0).

$$\begin{array}{rcl}
[\![\bullet]\!]_{\mathrm{Lit}} & : & \mathsf{Literal} \nrightarrow \mathbb{V} \\
[\![\bullet]\!]_{\mathrm{Conv}} & : & \mathsf{PrimType} \longrightarrow \mathbb{V} \nrightarrow \mathbb{V}
\end{array}$$

The function $[\![\bullet]\!]_{\text{Lit}}$ is undefined if one of its parameters is not properly initialised or undefined. Consider as an example, the addition operator: it is defined through $[\![\bullet]\!]_{\text{Lit}}$ to correctly handle operands of compatible types (e.g. adding an integer value with a real value returns a real value). These definitions are classic and well-known, but cumbersome to be fully defined.

Let consider now an expression taking place in a statement $\mathsf{stm} \in \mathsf{Stm}$ inside an operation $\mathsf{qop} \in \mathsf{QOpN}$. Notice first that expressions and statements are supposed to be well-typed. From the grammar defining expressions in Fig. 4.2, we notice that evaluating expressions supposes to be able to determine the target of an instance expression. As a consequence, the function $[\![\bullet]\!]$ for evaluating expressions' values and of the function $[\![\bullet]\!]_{\text{T}}$ for evaluating instance expressions' target are mutually recursive.

The target of a local name (either a variable or a parameter name, or **self**) is the local name itself. The target of property access is defined only if its associated target denotes a valid instance in the domain. In this case, the target is composed of the object corresponding to the instance expression's target (either **super** or a general $\mathsf{instExp}$) and of the property given by the expression. The class where the property is declared is computed based on the class returned by the type-checking of the target expression.

$$
\begin{array}{rcl}
[\![\bullet]\!]_{\text{T}} : \mathsf{InstExp} \longrightarrow \mathbb{D} & \nrightarrow & \mathbb{T} \\
[\![\textbf{self}]\!]_{\text{T}}(d) & \mapsto & \textbf{self} \\
[\![\mathsf{varN}]\!]_{\text{T}}(d) & \mapsto & \mathsf{varN} \\
[\![\mathsf{paramN}]\!]_{\text{T}}(d) & \mapsto & \mathsf{paramN} \\
[\![\textbf{super}.\mathsf{propN}]\!]_{\text{T}}(d) & \mapsto & (o, \mathrm{decl}((\mathsf{pkg}, \mathsf{c}'), \mathsf{propN})) \text{ if } o = d(\textbf{self}) \\
 & & \text{and } \tau \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \textbf{super.}\mathsf{propN} \rhd (\mathsf{pkg}, \mathsf{c}') \\
[\![\textbf{instExp}.\mathsf{propN}]\!]_{\text{T}}(d) & \mapsto & (o, \mathrm{decl}((\mathsf{pkg}, \mathsf{c}'), \mathsf{propN})) \text{ if } o = [\![\mathsf{instExp}]\!](d) \\
 & & \text{and } \tau \bullet \mathsf{qop} \vdash_{\mathsf{Expr}} \mathsf{instExp.propN} \rhd (\mathsf{pkg}, \mathsf{c}')
\end{array}
$$

The evaluation of the **null** expression and of literal expressions is straightforward, considering the dedicated function $[\![\bullet]\!]_{\text{Lit}}$. The value of the **instanceof** test depends on the value of its instance expression: it is true if the type of the instance expression specialises the class declaration. Notice that if this value is undefined, then the result is also undefined. The value of an instance expression is the value associated in the domain with the instance expression's target.

$$
\begin{array}{rcl}
[\![\bullet]\!] : \mathsf{Exp} \longrightarrow \mathbb{D} & \nrightarrow & \mathbb{V} \\
[\![\textbf{null}]\!](d) & \mapsto & \textbf{null} \\
[\![\mathsf{lit}]\!](d) & \mapsto & [\![\mathsf{lit}]\!]_{\text{Lit}} \\
[\![\mathsf{instExp}\ \textbf{instanceof}\ (\mathsf{pkg}\ \mathsf{c})]\!](d) & \mapsto & \mathrm{type}([\![\mathsf{instExp}]\!](d)) \preccurlyeq (\mathsf{pkg}, \mathsf{c}) \\
[\![\mathsf{instExp}]\!](d) & \mapsto & d([\![\mathsf{instExp}]\!]_{\text{T}}(d))
\end{array}
$$

Updating

The assignment statement is crucial to the behavior of the AL: it ensures that object integrity is preserved during updating. Since assignments can transparently update a variable, a reference, an association or a containment, its behavior must ensure the global consistency and the containment uniqueness property of models.

Suppose an assignment statement $\mathsf{lhs} = \mathsf{exp}$ evaluated in a domain $d \in \mathbb{D}$, for which expression exp evaluates to the value $v \in \mathbb{V}$, and left-hand side (LHS) lhs refers to the target $t \in \mathbb{T}\mathsf{arget}$ whose type is t (formally speaking, $[\![\mathsf{exp}]\!](d) = v$, $[\![\mathsf{lhs}]\!]_{\text{T}}(d) = t$ and $\lambda \bullet \mathsf{op} \vdash_{\mathsf{Expr}} \mathsf{lhs} \rhd \mathsf{t}$). The effect of the assignment depends on both the *type* and the *nature* of the LHS involved [14, Appendix A].

(i) $\mathsf{t} \in \mathsf{DataType}$ If the target's type is a $\mathsf{DataType}$, it does not deal with containment and t's value is simply replaced by v after properly converting it in case of numerical values.

$\mathsf{t} \notin \mathsf{DataType}$ If not, then t represents an object o and there is two cases:

(ii-1) $t \in \mathsf{LocalN}$ either t refers to a local name and the assignment has the usual effect of replacing the current object's value by the the object denoted by v;

(ii-2) $t \in \mathbb{O} \times \mathsf{PropN}$ or $t = (o, \mathsf{p})$ is a property access and it depends on the collection nature of p. The case without collection is depicted in Fig. 4.5: in this case, v is assigned to t, the container of the previous object $x \triangleq d(t)$ pointed by o is reset; and in case of bidirectional reference, the opposite property q is set to o to preserve consistency. If there is a collection, then this process is repeated to all object within the collection.

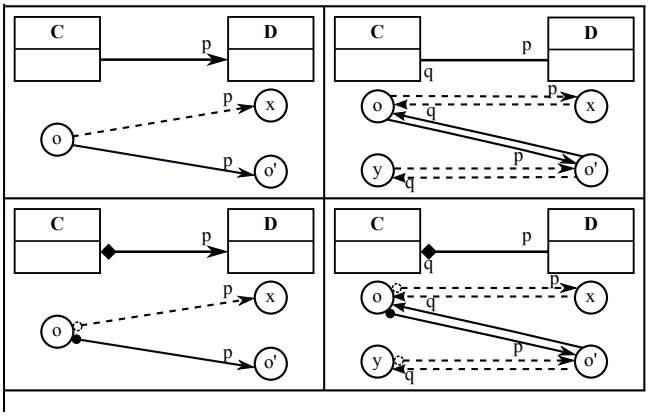


Figure 4.5: Assignment of objects in a reference. Dashed/plain arrows represent the situation before/after assignment. Plain circles denote current object's containers; empty ones a container's reset.

The definition of the assignment function $[\![\bullet, \bullet]\!]$ reflects these remarks and addresses the situation with collection values: here, the update is done on each target object t' of all objects contained within the collection value v, i.e. $t' \triangleq (o', \mathrm{opp}(p))\ \forall o' \in \mathrm{objs}(v)$, where the function $\mathrm{objs} : \mathbb{V} \to \wp(\mathbb{O})$ retrieves all objects contained in a collection value.

$$[\![\bullet, \bullet]\!] : \mathsf{Lhs} \times \mathsf{Exp} \nrightarrow \mathbb{D} \longrightarrow \mathbb{D}$$

$$(\mathsf{lhs}, \mathsf{exp})(d) \mapsto \begin{cases} d[t \mapsto [\![\mathsf{t}, v]\!]_{\mathrm{Conv}}] & \text{if (i)} \\ d[t \mapsto (v)] & \text{if (ii} - 1\text{)} \\ d\begin{bmatrix} t \mapsto v \\ t' \mapsto (\_, o) \\ x \mapsto (\_, \bot) \end{bmatrix} & \text{if (ii} - 2\text{)} \end{cases}$$

$$\mathrm{objs} : \mathbb{V} \longrightarrow \wp(\mathbb{O}) \qquad v \mapsto \begin{cases} v & \text{if } \mathrm{type}(v) \in (\_, \mathsf{PClassN}) \\ \varnothing & \text{otherwise} \end{cases}$$

4.4.2 Configuration

The set of *configurations* Γ consists in the label denoting the statement under execution, a stack storing the information between operation calls, and the semantic domain. A *stack* is a sequence whose elements comprise the label where to resume after completing the execution of a call, the local environment of the call, and eventually the variable to which the result of the call need to be assigned.

$$\begin{aligned} \Gamma &\triangleq \mathsf{Lab} \times \langle \mathbb{E}\mathsf{nv} \rangle \times \mathbb{D} \\ \mathbb{E}\mathsf{nv} &\triangleq (\mathsf{Lab} \times \mathbb{L}) \cup (\mathsf{Lab} \times \mathbb{L} \times \mathsf{VarN}) \end{aligned}$$

Suppose now that a Kermeta transformation is launched from the platform, i.e. a **main** operation $\mathsf{op}_{\mathsf{main}}$ inside a **main** class $\mathsf{C}_{\mathsf{main}} = (\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}})$ (cf. Appendix A) is called with correct effective parameter value list (v_{param}), where $\mathsf{param} \in \mathsf{Dom}\,(\lambda(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}, \mathsf{op}_{\mathsf{main}}))$. From this information, we build an *initial* configuration $\gamma_0 \in \Gamma$ that starts the execution, which consists in the following: the first label of $\mathsf{op}_{\mathsf{main}}$'s (non-empty) body, an empty stack environment, and a domain where the model m_0 contains only one object of type $\mathsf{C}_{\mathsf{main}}$ for which all properties are initialised to their default value, and a local environment where each parameter param is bind to the corresponding value v_{param} and each variable to its default value.

$$\gamma_0 = (\mathsf{lab}_0, \langle\,\rangle, (m_0, l_0)) \text{ with } \begin{cases} m_0 = \mathfrak{Initialise}(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}) \\ \forall \mathsf{param} \in \mathsf{Dom}\,(l_0), l_0(\mathsf{param}) = v_{\mathsf{param}} \\ \forall \mathsf{var} \in \mathsf{Dom}\,(l_0), l_0(\mathsf{var}) = \mathfrak{Default}(\mathsf{ct}_{\mathsf{var}}) \\ \text{with } \lambda(\mathsf{pkg}_{\mathsf{main}}, \mathsf{c}_{\mathsf{main}}, \mathsf{op}_{\mathsf{main}})(\mathsf{var}) = (\_, \_, \mathsf{ct}_{\mathsf{var}}) \\ \langle\!\langle \mathsf{lab}_0, \ldots, \mathsf{lab}_n \rangle\!\rangle = \mathrm{labs}((\mathsf{pkg}, \mathsf{c}, \mathsf{op})) \end{cases}$$

4.4.3 Semantic Rules

The rewriting system takes the form of a rule "$\gamma \xrightarrow{\mathsf{stm}} \gamma'$", meaning that the configuration $\gamma \in \Gamma$ is rewritten in γ' when executing the statement $\mathsf{stm} \in \mathsf{Stm}$ under some conditions. All operations that occur in a rule are considered atomic. Notice that inside the rules, we explicitly recall the label to bind the statement with the control flow.

Conditional Statement

A conditional statement CondStmt only changes the label to the adequate one, according to the boolean value of its expression.

$$\frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ (\mathsf{lab}', \mathsf{lab}'') & = & \mathrm{nxt}(\mathsf{lab}) \\ v & \Longrightarrow & \mathsf{lab}_{\mathsf{res}} = \mathsf{lab}' \\ \neg v & \Longrightarrow & \mathsf{lab}_{\mathsf{res}} = \mathsf{lab}'' \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ if\ exp}} (\mathsf{lab}_{\mathsf{res}}, S, d)}$$

Instance Creation Statement

An instance creation statement NewInstStmt adds to the model a fresh initial object, and associates it to the variable in the local environment. The execution proceeds to the next label.

$$\frac{\begin{array}{rcl} o & \notin & \mathsf{Dom}\,(m) \\ m' & = & m \cup (o \mapsto \mathfrak{Initialise}(\mathsf{pkg}, \mathsf{c})) \\ l' & = & l[\mathsf{var} \mapsto o] \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\mathsf{lab:\ var\ =\ \textbf{new}\ (pkg\ c)}} (\mathsf{lab}', S, (m', l'))}$$

Assignment and Cast Statements

An assignment AssignStmt has to preserve the integrity of objects. Since we already defined a function for this purpose, the rule simply applies it and proceeds to the next label.

$$\frac{\begin{array}{rcl} (m', l') & = & [\![\mathsf{lhs}, \mathsf{exp}]\!](m, l) \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\mathsf{lab:\ lhs\ =\ exp}} (\mathsf{lab}', S, (m', l'))}$$

A cast statement CastStmt essentially behaves like an assignment: it assigns the right expression to the variable, except that it has to carefully handle the dynamic types. In the case of primitive values, a cast statement simply behaves as a conversion. In the case of instances, the right-hand side's dynamic type must specialise the left-hand side's static type.

$$\frac{\begin{array}{rcl} \mathsf{qop} & = & \mathrm{op}(\mathsf{lab}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{varN} \rhd \mathsf{t} \in (\bot, \mathsf{PrimType}) \\ v & = & [\![\mathsf{exp}]\!](d) \\ d' & = & [\![\mathsf{varN}, [\![\mathsf{t}]\!]_{\mathrm{Conv}}(v)]\!](d) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ varN?=exp}} (\mathsf{lab}', S, d')} \qquad \frac{\begin{array}{rcl} \mathsf{qop} & = & \mathrm{op}(\mathsf{lab}) \\ \lambda \bullet \mathsf{qop} & \vdash_{\mathsf{Expr}} & \mathsf{varN} \rhd \mathsf{t} \in (\bot, \mathsf{PClassN}) \\ v & = & [\![\mathsf{exp}]\!](d) \\ \tau(v) & \leqslant & \mathsf{t} \\ d' & = & [\![\mathsf{varN}, v]\!](d) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\mathsf{lab:\ varN?=exp}} (\mathsf{lab}', S, d')}$$

Operation Management

We include in this Section three kind of statements: the operation calls, the return statement and the collection statements. The two first manipulate the stack, whereas the last only delegates the execution to predefined operations of the semantic domain.

A return statement ReturnStmt proceeds to the label stored in the top element of the stack and changes the current local environment with the stored one, then removes the top element. If the return statement has an expression, it is evaluated in the context of the stored local environment and assigned to the variable stored in the top element.

$$\frac{s = (\mathsf{lab}', l')}{(\mathsf{lab}, s :: S, (m, l)) \xrightarrow{\text{lab: } \textbf{return}} (\mathsf{lab}', S, (m, l'))}$$

$$\frac{\begin{array}{rcl} s & = & (\mathsf{lab}', l', \mathsf{var}) \\ (m', l'') & = & [\![\mathsf{var}, \mathsf{exp}]\!](m, l') \end{array}}{(\mathsf{lab}, s :: S, (m, l)) \xrightarrow{\text{lab: } \textbf{return exp}} (\mathsf{lab}', S, (m', l''))}$$

An operation call without assignment of the form $\mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$ proceeds as follows. First, the target's value is computed and ensured by the type-checking system to denote an instance. Then, all effective parameters' values are computed. Then, the operation's body is looked up, based on the dynamic type of the target. The $Start$ function then builds the new domain based on the parameters values and the old domain. The label where to continue the execution is saved in the stack as well as the current environment.

$$\frac{\begin{array}{rcl} v_{\mathsf{this}} & = & [\![\mathsf{target}]\!](m, l) \\ \forall i,\ v_i & = & [\![\mathsf{exp_i}]\!](m, l) \\ (\bot, (\mathsf{pkg}, \mathsf{c})) & = & \tau(v_{\mathsf{this}})(m, l) \\ (\_, params, \_, b) & = & \omega(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) \\ (m', l') & = & \mathfrak{Start}(\mathsf{pkg}, \mathsf{c}, \mathsf{op})(v_{\mathsf{this}}, \langle\!\langle v_1, \ldots, v_n \rangle\!\rangle)(m, l) \\ (m', l') & = & [\![\mathsf{lhs}, \mathsf{exp}]\!](m, l) \\ \langle\!\langle \mathsf{lab}_0, \ldots \rangle\!\rangle & = & \mathrm{labs}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \\ (\mathsf{lab}', l) & = & s_0 \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\text{lab: } \mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)} (\mathsf{lab}_0, s_0 :: S, (m', l'))}$$

An operation call with assignment of the form $\mathsf{varN} = \mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)$ proceeds the same as previously, except that it has to save the variable into the store.

$$\frac{\begin{array}{rcl} v_{\mathsf{this}} & = & [\![\mathsf{target}]\!](m, l) \\ \forall i,\ v_i & = & [\![\mathsf{exp_i}]\!](m, l) \\ (\bot, (\mathsf{pkg}, \mathsf{c})) & = & \tau(v_{\mathsf{this}})(m, l) \\ (\_, params, \_, b) & = & \omega(\mathsf{pkg})(\mathsf{c})(\mathsf{op}) \\ (m', l') & = & \mathfrak{Start}(\mathsf{pkg}, \mathsf{c}, \mathsf{op})(v_{\mathsf{this}}, \langle\!\langle v_1, \ldots, v_n \rangle\!\rangle)(m, l) \\ (m', l') & = & [\![\mathsf{lhs}, \mathsf{exp}]\!](m, l) \\ \langle\!\langle \mathsf{lab}_0, \ldots \rangle\!\rangle & = & \mathrm{labs}(\mathsf{pkg}, \mathsf{c}, \mathsf{op}) \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \\ (\mathsf{lab}', l, \mathsf{varN}) & = & s_0 \end{array}}{(\mathsf{lab}, S, (m, l)) \xrightarrow{\text{lab: } \mathsf{varN} = \mathsf{target.op}(\mathsf{exp}_1, \ldots, \mathsf{exp}_n)} (\mathsf{lab}_0, s_0 :: S, (m', l'))}$$

A collection statement CollStmt first computes the collection corresponding from the call expression and the updating element from the parameter expression, then updates the collection accordingly to the native statement and proceeds to the next statement. Notice that the type checking system ensures that the element can effectively update the collection, since its type specialises the collection's type.

$$\frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ v' & = & [\![\mathsf{exp}']\!](d) \\ d' & = & d[v \mapsto v \oplus v'] \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\text{lab: } \mathsf{exp}.\textit{add}(\mathsf{exp})} (\mathsf{lab}', S, d)}$$

$$\frac{\begin{array}{rcl} v & = & [\![\mathsf{exp}]\!](d) \\ v' & = & [\![\mathsf{exp}']\!](d) \\ d' & = & d[v \mapsto v \ominus v'] \\ (\mathsf{lab}', \_) & = & \mathrm{nxt}(\mathsf{lab}) \end{array}}{(\mathsf{lab}, S, d) \xrightarrow{\text{lab: } \mathsf{exp}.\textit{del}(\mathsf{exp})} (\mathsf{lab}', S, d)}$$

4.5 Discussions

This Section starts by discussing some syntactic aspects of Kermeta's AL in comparison with our core AL, and then compares with related works in the domain of ALs in particular and model transformation in general.

4.5.1 Syntactic aspects & Restrictions

Returning from an operation's body (`result`). As the running example presented in Sec. 2.2 and the way it is transformed in Fig. 4.3 suggest, Kermeta's keyword `result` is equivalent in our formalisation to **`return`**.

File Dependency (`require`). We did not take into account the way Kermeta deals with file dependency since it is a pure static consideration: it only affects how a complete Kermeta specification is obtained, which can be handled by adequate pre-transformations.

Derived Properties are in MOF a special kind of property whose value is computed from other properties. As a matter of fact, it is not *per se* a concrete element as it depends from others. Derived properties value usually depends on other properties' value: Kermeta allows to attach a `getter` for retrieving the value, and eventually a `setter` to modify it, so that their value is automatically updated if other values they depend on are. We did not address this feature, since it is possible to fully mimic this capacity by using normal operations (which is the way they are implemented in).

Values Casting. Kermeta possesses two constructions for value casting. The *conditional cast statement* **`:=`** is part of our AL (cf. Sec. 4.2.2 and 4.4). Our semantics first detects numerical values casting (which are statically decidable) from instance casting that sometimes can fail. Kermeta assigns in this case the special value **`void`** (or equivalently in our AL, **null**) that is not taken into account because the AL does not handle exceptions and the semantic rules only rewrite correct configurations. Notice also that because primitive types are meta-represented in Kermeta (cf. Discussion Sec. 3.7), only the second rule on instances matter; the first was given for soundness and completeness purpose.

Kermeta also proposes the `asType()` operation (cf. [12, §2.9.2.2]) that avoids using an intermediate variable. The only difference resides in the error cases handling: when impossible to cast, `asType` raises an exception instead of silently failing by assigning `void`.

Collection API. Without a proper representation of generic classes and associated functionals, we equipped our core AL with only the basic capabilities for manipulating collections: the empty testing the emptiness or computing the size, adding/removing an element, and accessing an element at a given index within an ordered collection. These constructions give the idea for extending the core AL with more sophisticated constructions, based on the basic operations on Abstract Datatypes defined in 2.3.2: basic constructions' semantics only consists of directly mapping it to the corresponding operator, whereas other ones can be defined in terms of the basic ones.

The Collection API also defines specialised cast operations of the form `asXxx()` (where `Xxx` stands for the name of a collection, e.g. `asSequence()`). These operations are the counterpart of the downward/upward conversion operators of Def. 2.4.

4.5.2 Related Works

Two transformation languages are directly comparable to Kermeta's AL: the MOF Action Language [24] and xOCL [9]. They include OO features as well as queries and more complicated behavior like State Machines, but to the best of our knowledge, a formal specification in terms of Action Semantics only exists for UML [36]. Approaches like ASM and GB take advantage of the target workspace to perform the transformations, which require from the user knowledge about it (cf. the work of Combemale *et al.* in [11] for a comprehensive review). Our approach contrast with these by formally specifying the framework itself, making formal any DSML whose semantics is expressed with Kermeta directly in the Kermeta workspace, instead of relying on target languages. Fleurey outlined in his pioneering work about Kermeta [14] almost the same AL subset as ours. Nevertheless, his work is questionable in several outcomes: the structuring concepts used for the AL lacks a formal counterpart; he uses a big-step semantics, which is not directly executable; and his AL subset lacks a formal type system.

Chapter 5

Implementation & Applications

As we claimed before, specifying a language's semantics is not a finality on itself. One of the first application of such a specification is to serve as a reference for potential implementations [11] and as a clear reference documentation for engineers, researchers and more generally, any user of the language.

Our formalisation has several strengths because of its mathematical nature. First, it is expressed only with the classical computer science mathematical formalisms which are usually part of engineers' and researchers' background, making it easier to understand without learning a specific syntax. Second, it is high level and acts like an algorithm regarding its implementations, meaning that it expresses the essence of Kermeta's semantics but without relying on particular features that could have facilitate its expression. As a consequence, depending on how this semantics is used, one can easily take advantage of for example, a native implementation of collections or primitive types, or an already object-oriented language. But the very same reason can be regarded as a drawback: being mathematical, this specification cannot be directly exploited.

This Chapter starts by first describing in Sec. 5.1 our own experience with concrete implementations of Kermeta's semantics, used to give us confidence of its correctness and to validate the approach. Then, for illustrating the fact that a semantics specification is a required step before several concrete DSLs activities, we discuss some concrete applications in Sec. 5.2.

5.1 Concrete Implementation

When specifying a formal semantics from an informal description, one has always to deal with ambiguities and under-specifications. Many discussions with Triskell Team members as well as several experiments on Kermeta's platform itself helped us to apprehend and gain insight about Kermeta's languages. For example, extending the from clause already used for operations (where the *name* role is played by operation's *signature*) was a discussed natural simple choice required in order to have a full static type-checking.

Aside from this Kermeta-specific construction, the structural part's specification is directly adaptable to any MOF-like language. To further enforce our confidence, we implemented it in Z [**Spivey:1992aa**] together with a small example that covers all structural constructions. It took approximatively one week for an expert to read, understand and implement the specification and the example, and three days to fully achieve the conformance proof [2].

Several lessons were learned. First, we have chosen Z because we knew the gap between its language and the "plain math" specification is small. Second, as expected, implementing some parts of the specification described with classical computer science formalisms irremediably suffer from the accidental complexity of the chosen tool: for example, Z lacks a proper specification for strings and we therefore used named identifiers, with the extra burden of adding constraints to ensure names' unicity; the type algebra (corresponding to MType) required the use of *free types*, which are difficult to manipulate when writing the example by hand. Some structural components, like coupled opposite references, were better represented in a way that does not strictly follow the specification to take advantage of Z relation definitions as well as to overcome the well-formedness constraints specific to attributes and references. Finally, Z/EVE, the associated prover we used, does not offer specialised proof tactics and strategies that are a key point for the conformance proof that heavily relies on definition expansions; we think we can benefit from more specialised theorem-provers.

Currently, we are exploring tools for implementing the AL, that offer a natural way for expressing rewriting systems together with formal verification capabilities. MAUDE [10] appears to be a good candidate: its formal foundations adequately meets the specification's requirements by natively supporting sorts (used by the grammar and many of the

semantic operations), equational theories (adequate to fully implement recursive constructions) and rewriting (both for the type-checking system and the rewriting rules). Furthermore, several implementations of the structural part, closed to our Z implementation, already exist (cf. [5, 28]) and MAUDE, which will save us a precious time.

5.2 Applications

Metamodels and models, used in the context of DSLs, are at the core of the MDE approach. Their formal undergrounds handled historically structural design, but have now to focus on behaviour to scale up and address formal verification, a necessary step when dealing with safety critical and embedded applications. As a first step towards formal verification, we contributed a full formal specification of Kermeta, a metamodeling framework equipped with an object oriented action language, widely used in the MDE arena. This specification is expressed with mathematical formalisms instead of using a tool syntax: it makes it easier to understand, and serves as a basis for potential implementations or translations in dedicated verification tools. The specification is also modular: it provides a set-theoretical semantics to Kermeta Structural Language that can be reused for any MOF-like metamodeling language; and expresses Kermeta Action Language using structural operational semantics. Currently, these semantics are under implementation in Maude, which natively support execution from formal specification, and offers a large variety of verification techniques such as model-checking and theorem-proving. We then want to investigate dynamic properties of model transformations, which is a crucial concern in critical systems.

Chapter 6

Conclusion

This paper addressed the formal specification of an important subset of Kermeta, an widely used metamodeling framework for the specification of DSMLs. The formalisation uses a mathematical framework independent of any tool syntax to clearly formalise, and clearly separate the material for the Structural Language, that can be used for any MOF-like language, from the Action Language specific to Kermeta. As such, this work can serve as a reference formal description from which implementation into specific tools can be derived, and be used as a common description for comparing implementations. Future works will be follow two directions: adding important missing features of Kermeta (model type and aspects); and exploring formal verification techniques derived from this semantics specification.

Acknowledgments. This research is partially funded by the Luxembourgish FNR (Fonds National de la Recherche). The authors would like to warmly thank Benoît Combemale and Didier Vojtisek for discussions and technical support about Kermeta, and early comments on this work.

Appendix A

The Finite State Machine Example

```
package FSM;
    require kermeta
    using kermeta::standard

    enumeration Kind {NORMAL;START;STOP;}

    class Label{
        attribute label: String
    }


     // FSM assumes there is only one START and one FINAL State
    class FSM inherits Label{
        attribute alphabet: set String [1..*]
        reference states: seq State [1..*] # fsm
        reference transitions: seq Transition [0..*]# fsm

        operation getStart(): State is do
            var i : Integer init 0
            from i := 0until
                i == states.size() or states.at(i).kind == Kind.START
            loop
                i := i+1
            end
            if i == states.size() then
                result := void
            else
                result := self.states.at(i)
            end
        end

        operation getFinal(): State is do
            var i : Integer init 0
            from i := 0 until
                i == states.size() or states.at(i).kind == Kind.STOP
            loop
                i := i+1
            end
            if i == states.size() then
                result := void
            else
                result := self.states.at(i)
            end
        end

        operation accept(word: seq String [0..*]) : Boolean is do
            var current: State init self.getStart()
            var final : State init self.getFinal()
            var toEval : seq String[0..*] init word
            var isNull : Boolean init false

            from var i : Integer init 0 until
                i == toEval.size() or isNull
            loop
                current := current.fire(toEval.at(i))
                if(current.isVoid) then
                        isNull := true
                end
                i := i+1
            end
            result := (current == final)
        end
```

```
}

 class State inherits Label{
    attribute kind: Kind
    reference fsm: FSM # states
    reference in: Transition [0..*] # tgt
    reference out: Transition [0..*] # src

    operation fire(letter: String): State [0..1] is do
        var trans: seq Transition [0..*] init self.out.asSequence()

        if(trans.isVoid) then
            result := void
        else
            var current: Transition init trans.at(0)
            from var i : Integer init 0 until
                i == trans.size() or trans.at(i).label == letter
            loop
                i := i+1
            end
            if(current.isVoid) then
                result:= void
            else
                result:= current.tgt
            end
        end
    end
}


 class Transition inherits Label{
    reference fsm: FSM # transitions
    reference tgt: State [1..1] # in
    reference src: State [1..1] # out
}
```

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques & Tools*. Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[2] Moussa Amrani and Nuno Amálio. *A Set-Theoretic Formal Specification of the Semantics of Kermeta*. Tech. rep. Available at `http://bit.ly/ju4NcG`. University of Luxembourg, 2011.

[3] Colin Atkinson and Thomas Kühne. "Model-Driven Development: A Metamodeling Foundation." In: *IEEE Software* 20 (2003), pp. 36–41. ISSN: 0740-7459. DOI: `http://doi.ieeecomputersociety.org/10.1109/MS.2003.1231149`.

[4] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Precise Semantics of EMF Model Transformations by Graph Transformation." In: *Models'08*. Toulouse, France, 2008, pp. 53–67. DOI: `http://dx.doi.org.proxy.bnl.lu/10.1007/978-3-540-87875-9_4`.

[5] Artur Boronat and José Meseguer. "An Algebraic Semantics For MOF." In: *FASE'08*. Springer, 2008, pp. 377–391.

[6] Frederick P. (Jr.) Brooks. "No Silver Bullet – Essence and Accidents of Software Engineering (Invited Paper)." In: *IFIP 10th World Computer Congress*. Dublin, Ireland, 1986, pp. 1069–1076.

[7] Luca Cardelli. "Type Systems." In: *The Computer Science and Engineering Handbook*. CRC Press, 2004.

[8] Giuseppe Castagna and Zhiwu Xu. "Set-Theoretic Foundation of Parametric Polymorphism and Subtyping." In: *16th ACM SIGPLAN* ICFP. 2011.

[9] Tony Clark, Paul Sammut, and James Willans. *Superlanguages: Developing Languages and Applications with* XMF. Ceteva, 2008.

[10] Manuel Clavel et al. *All About* MAUDE. *A High-Performance Logical Framework*. Springer, 2007.

[11] Benoit Combemale et al. "Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification." In: *Journal of Software* 4.9 (November 2009), pp. 943–958.

[12] Zoé Drey et al. *Kermeta Language — Reference Manual*. University of Rennes, Triskell Team. 2009.

[13] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications*. Springer-Verlag, 1985.

[14] Frank Fleurey. "Language and Method for Trustable Modeling Engineering." (in french). Doctoral dissertation. University of Rennes (France), 2006.

[15] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "A Semantic Framework for Metamodel-Based Languages." In: *Automated Software Engineering (*ASE*)* 16.3–4 (2009), pp. 415–454.

[16] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "Model-Driven Language Engineering: The ASM-META Case Study." In: *ICSEA '08*. 2008, pp. 373–378. ISBN: 978-0-7695-3372-8. DOI: `http://dx.doi.org/10.1109/ICSEA.2008.62`.

[17] David Harel and Bernhard Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10 (2004), pp. 64–72. ISSN: 0018-9162. DOI: `http://doi.ieeecomputersociety.org/10.1109/MC.2004.172`.

[18] Stefan Jurack and Gabriele Taentzer. "A Component Concept for Typed Graphs With Inheritance and Containment Structures." In: *5th Proceedings of ICGT*. 2010.

[19] Kermeta. Triskell Team (University of Rennes, France) `http://www.kermeta.org`.

[20] P. Krishnan. "Consistency Checks For UML." In: *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (*APSEC*)*. 2000.

[21] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. “Weaving Executability into Object-Oriented Meta-Languages.” In: *MODELS/UML’2005*. 2005, pp. 264–278.

[22] Object Management Group. *Meta-Object Facility 2.0 Core Specification (06-01-01)*. Tech. rep. OMG, 2006.

[23] Object Management Group. *Object Constraint Language (*OCL*) Specification (Version 2.2, formal/2010-02-01)*. Specification Document. (OMG), 2010.

[24] Richard Paige, Dimitrios Kolovos, and Fiona Polack. “An Action Semantics for MOF 2.0.” In: ACM SAC. Dijon, France, 2006, pp. 1304–1305. ISBN: 1-59593-108-2. DOI: http://doi.acm.org/10.1145/1141277.1141579. URL: http://doi.acm.org/10.1145/1141277.1141579.

[25] Iman Poernomo. “The Meta-Object Facility (MOF) Typed.” In: *SAC*. 2006, pp. 1845–1849.

[26] Isabelle Pollet. “Towards a Generic Framework For the Abstract Interpretation of Java.” Doctoral dissertation. Catholic University of Louvain (Belgium), 2004.

[27] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. “Towards A Rigorous Semantics Of UML Supporting Its Multiview Approach.” In: *Fundamental Approaches to Software Engineering (*FASE*)*. 2001.

[28] José Eduardo Rivera. “On The Semantics of Real-Time Domain-Specific Modeling of Languages.” Doctoral dissertation. University of Malaga (Spain), 2010.

[29] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation (Vol. I: Foundations)*. World Scientific Pub., 1997. ISBN: 98-102288-48.

[30] Dan Song et al. “A Formal Language For Model Transformation Specification.” In: *In Proceedings of 7th* ICEIS. 2005.

[31] Kim Soon-Kyeong and David Carrington. “Formalizing the UML Class Diagram Using Object-Z.” In: *Conference on* UML. 1999.

[32] Robert F. Stark, E. Borger, and Joachim Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. ISBN: 3540420886.

[33] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[34] TopCased. The Open-Source Toolkit for Critical Systems http://www.topcased.org.

[35] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[36] M. Yang, G.J. Michaelson, and R.J. Pooley. “Formal Action Semantics for a UML Action Language.” In: *Journal of Universal Computer Science* 14.21 (2008), pp. 3608–3624.