Formally Defining and Iterating Infinite Models ⋆

Benoit Combemale[1], Xavier Thirioux[2], and Benoit Baudry[3]

[1] University of Rennes 1, IRISA, France
[2] INPT ENSEEIHT, IRIT, France
[3] Inria, Centre Rennes Bretagne Atlantique, France

Abstract. The wide adoption of MDE raises new situations where we need to manipulate very large models or even infinite model streams gathered at runtime. These new uses cases for MDE raise challenges that had been unforeseen by the time standard modeling framework were designed. This paper proposes a formal definition of an infinite model, as well as a formal framework to reason on queries over infinite models. This formal query definition aims at supporting the design and verification of operations that manipulate infinite models. First, we precisely identify the MOF parts which must be refined to support infinite structure. Then, we provide a formal coinductive definition dealing with unbounded and potentially infinite graph-based structure.

1 Introduction

The growing adoption of Model-Driven Engineering (MDE) at all steps of software development comes with new requirements for MDE theories and tools. In particular, this work focuses on the need to process (i) finite but very large models, and (ii) infinite models. A major challenge to process these categories of models consists in understanding the exact meaning of a query over a model for which the interpretation does not know the size at a given point in time.

To illustrate the need to process finite but very large models, let us consider the complete model representing the entire Eclipse platform (the minimal workbench with OSGi). This model includes about 5 million model elements. Current model processing tools require all the model elements in memory (*e.g.*, Eclipse Modeling Framework (EMF) [1]). With EMF, the model of the Eclipse platform Java code requires 900MB in RAM memory. Steel et al. [2] provide an even bigger example: when they adopted a MDE approach to analyze civil engineering models, they had to deal with more than 7.3 million computational objects. Programming languages provide a good source of inspiration to deal with these issues. Through the notion of lazy evaluation, programming languages allow (lazy) iterations on potentially infinite data-structures. Even in Java, which does not support laziness natively, skilled programmers tend to manually postpone object instantiations as much as possible, *i.e.* only when needed, in order to reduce instantaneous memory consumption. Recent work were inspired by this approach to propose lazy model transformations to process very large models [3], or NoSQL-based approaches for model persistence [4].

⋆ This work is partially supported by the EU FP7-ICT-2009.1.4 Project N°256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

Beyond the problem of very large models, stands the issue of processing infinite models. This requirement becomes more and more critical with the growing adoption of the *models@runtime* paradigm [5]. For instance, if we consider a monitoring autonomous system which relies on a model at runtime to abstract information from a complex event processing (the CEP) engine. The CEP will indefinitely provide information about the environment and thus one cannot consider that the model at runtime will be bounded as it could grow indefinitely. Another illustration can be found in the realm of reactive systems, which are modeled by transition systems intended to run forever. In this case, the model at runtime that records the trace of states and events triggered during execution is another form of infinite model. To deal with infinite models, we could leverage mechanisms established in the area of web feeds, such as RSS syndication. In these cases, data following a predefined format is timely and infinitely appended to an initial model (even though the RSS file usually corresponds to a sliding window because older elements are removed). However, as far as we know, no model processing solution adopts the notion of sliding window over an infinite flow of model elements in order to deal with infinite models.

If we look at the current state of MDE theories and tools to deal with large and infinite models, we make two observations. First, metamodeling formalisms and most of the tools are deeply rooted on the assumption that models include a bounded number of model elements and that this bound is known when computing a query over the model. Second, there exist some ad-hoc implementations to deal with these issues, but there is no formal definition of infinite models and no reference formal semantics for a query over an infinite model. The consequence of these two observations is that it is currently impossible to formally verify operations that process very large or infinite models. This is a major challenge for the adoption of MDE in the use cases discussed above.

In this paper, we tackle the two limitations listed above through two major contributions.

- a detailed analysis of current metamodeling standards and a precise identification on how they prevent the definition of infinite models. In order to face this, we propose an extension to the MOF formalism to enable the local definition of an infinite part of a model.
- a coinductive semantics for a query operation over an infinite part of a model. This semantics relies on a formal definition of infinite models, and provides both a reference for various implementations and the foundations for the verification of operations that must process models of unknown size.

The paper is organized as follows. Section 2 illustrates through a concrete example how the current metamodeling formalisms such as MOF prevent the definition and manipulation of infinite models. Based on this observation, Section 3 introduces MOF extensions supported by a formal definition of infinite models, and Section 4 proposes a coinductive semantics supporting the manipulation of such infinite models. Since the proposed formal semantics is independent of any implementation choice, we discuss in Section 5 the various existing and possible implementations of coinductive operators. Finally, we conclude and outline our perspectives in Section 6.

2 Illustrative Example: How MOF Does Not Support Infinite Models

Model Driven Engineering (MDE) considers software artifacts as abstract typed graphs (i.e., models conforming to precisely defined metamodels). As discussed in the introduction, we have to deal with increasingly large models. In many cases these models may even be considered of unbounded and infinite size (i.e., their size is *a priori* unknown). Since models are conforming to metamodels, such situations must be considered in the definition of metamodels. These metamodels are themselves implemented using a meta-language, usually compliant with the *Meta Object Facility* (MOF) [6], such as Ecore [1].

This section illustrates how a meta-language such as MOF ties MDE practices to a vision of finite and bounded models and thus prevents the definition and manipulation of infinite models. We illustrate these issues with the UML2 State Machine formalism.

2.1 UML2 State Machine as an illustrative example

The state machine sketched in the bottom left corner of Figure 1 conforms to the State Machine metamodel displayed in the middle left of Figure 1 (see the metamodel level). This metamodel defines a *StateMachine* as composed of several *State* elements, including an initial state, as well as several *Event* elements which it may react to. States are pairwise linked through *Transition* elements as *source* and *target* states. Each transition is triggered by a set of events (*Trigger*) and in return sends events (*SendEvent*) as the *effect* of its firing.

The execution semantics of such a state machine processes as follows: The first *RuntimeEvent* to be processed (that is, in our case, an *InjectEvent* element) is popped from the *eventToProcess* queue belonging to the state machine. This element represents an *EventOccurrence* of some event. A *RuntimeEvent* is either locally raised by the state machine transitions (*endogenous*) or brought by the environment (*exogenous*) and several other preceding events, which have been previously processed, constitute the *cause* for which it occurs. Runtime events keep on being popped and put aside as unhandled, until this set contains enough events, for some *outgoing* transitions of the current state to be triggered. Then, actually firing the transition pushes some new events at the end of the event queue, changes the current state of the machine and removes the triggering events from the unhandled ones.

Since a model conforms to a precisely defined metamodel, the underlying model of a state machine (graph of objects) follows the constraints expressed in the metamodel, according to the MOF semantics.

2.2 Bounded collections of properties

As illustrated in Figure 1, the upper bound of the collection *eventToProcess* is relative to its cardinality. In MOF, this cardinality is reified in terms of the `lower` and `upper` attributes of the `Property` construct (cf. top of Fig. 1). According to the OMG MOF Specification, the value of the `upper` attribute is typed by `UnlimitedNatural`

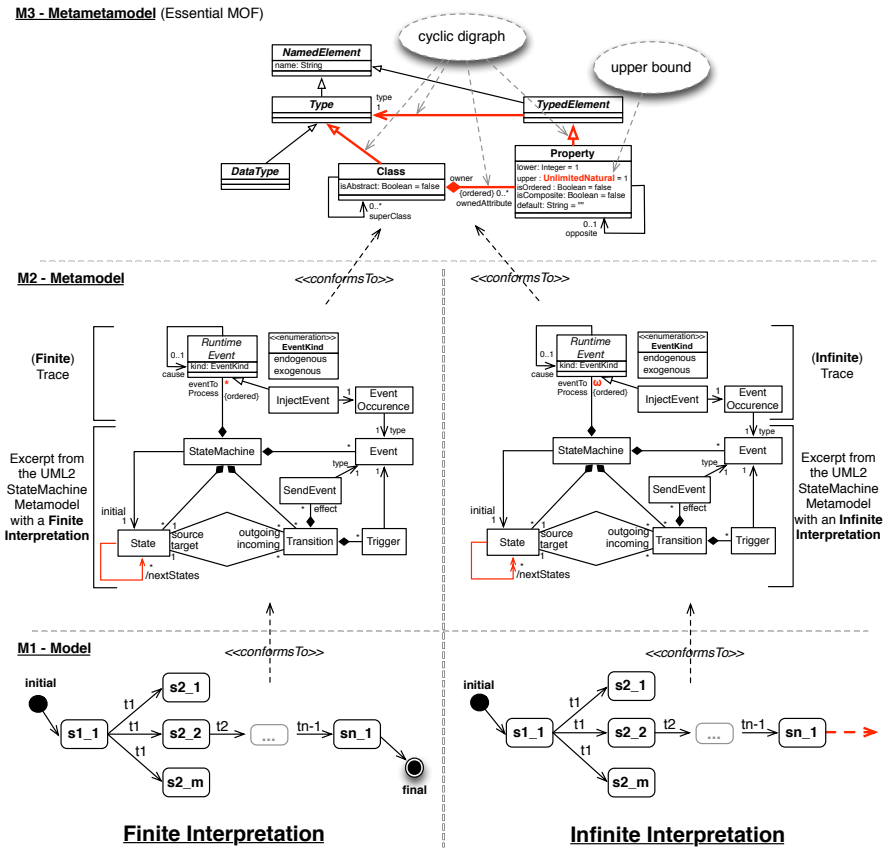

Fig. 1. Finite Model *vs.* Infinite Model

and must be of kind `LiteralUnlimitedNatural` taken from the UML's Kernel [6, §12.4 and §14.4]. The UML's kernel leaves open the concrete semantics implementation but involves a notation for the unlimited value (`*`) which *"denotes unlimited (and not infinity)"* [7, §9.11.7]. In the same specification, *unlimited* is reasonably interpreted as bounded (i.e., finite) in the type *Collection* used for the resulting collections, for instance by navigating through relationships. Indeed *"the semantics of the collection operations is given in the form of a postcondition that uses the* `IterateExp` *of the* `IteratorExp` *construct."* [8, §11.6.1]. Its execution semantics, which refers to `IterateExpEval`, is explicitly bounded in the specification [8, §10.3.2.14]. Consequently, the execution semantics of the iterators (*e.g.,* the ones coming from OCL) on the collection *eventToProcess* is bounded. This means that the iterators assume that all elements are considered as available at any time of the iteration.

2.3 Transitive closure

The underlying cyclic directed graph structure of any MOF-based metamodel (cf. top of Fig. 1) raises the issue of evaluating the transitive closure of a cycle. Such an issue can be shown in Figure 1, with the execution semantics of the closure over the next states (obtained from a given state with $outgoing \rightarrow collect(target)$). Since OCL 2.3, the standard includes a closure operation [8, §7.7.5]. This is very useful to specify recursive OCL operations. For instance, in Figure 1, the *reachable states* from a given state can be specified as proposed in the following OCL expression.

context State :: reachableStates () : **Set**(State) body :
{ self }−>closure(outgoing−>**collect**(target));

As stated by the OCL specification for the operator `closure`, *"the collection type of the result collection is the unique form (Set or OrderedSet) of the original source collection. If the source collection is ordered, the result is in depth first preorder."*. Here again, the underlying semantics refers to the type *Collection*. Consequently, the execution semantics of the closure is a finite processing, which assumes that the whole model is available for evaluation.

2.4 Discussion

Iterating both over the collection *eventToProcess* or the corresponding closure of the *reachable states* is thus a finite process for which the whole model is required (*e.g.,* a state machine defined at design time to model the behavior of a given class). The collection *eventToProcess* is bounded by the semantics of the attribute `upper` of a MOF property, and sets the "width" (i.e., number of outgoing edges from the same node) of the underlying graph of a conforming model. The *reachable states* process is finite, as defined by the underlying unfolding semantics as considered in the OCL operator `closure`, and sets the "depth" (i.e., length of a path with unique nodes) of the underlying graph of a conforming model.

These are strong limitations imposed by current metamodeling formalisms since, as illustrated in the introduction, such a state machine should also be considered as locally infinite (*e.g.,* state machine continuously updated to monitor at runtime a running and non-terminating program). In the following sections, we introduce slight modifications in MOF, which broaden its scope. In the context of infinite models, these modifications support the definition of a formal semantics for the MOF attribute `upper` and the unfolding semantics as used in the OCL operator `closure`.

3 Defining Infinite Models

This section starts with two proposals to extend the scope of MOF and allow the identification in metamodels of the infinite parts of the conforming models (i.e., parts which need to be manipulated despite their unknown size). These extensions are then used to provide a formal definition of infinite models.

3.1 Intuitive presentation

While the semantics described by the OMG in the MOF specification involves a finite interpretation of models, some situations require an infinite interpretation of the same structure. For instance, a state machine can be considered as infinite (cf. right part of Fig. 1) if it abstracts the execution trace of a non-terminating program. In practice, this execution trace can be lazily built at design time while exploring the graph of reachable states, or continuously built at runtime during the system execution (*e.g.,* monitoring).

As seen in the previous section and illustrated in the example depicted in Figure 1, a model can be infinite in two situations, respectively in *the width* and in *the depth* of the underlying graph. In the following, we come back into these two situations and we propose MOF extensions with a concrete syntax to locally characterize in a metamodel the infinite parts of the conforming models.

- The collection defined in Figure 1 by the relation eventToProcess on StateMachine may be considered as infinite in case of a non terminating execution (*i.e.,* an infinite sequence of runtime event). We are noting ω the upper bound of the multiplicity of an infinite collection (cf. right part of Fig. 1), compared to the $*$ value which defines an unbounded but finite collection (cf. left part of Fig. 1). We assume that an infinite collection is ordered and then countable.
- A reflexive relation may be indefinitely unfolded and then, the computation of the closure may not terminate (currently, common practices consider that the closure computation terminates). This situation may even be mandatory if we consider in this relation a multiplicity with a lower bound greater than zero. We propose to graphically note such reflexive relations with infinite unfoldings as an arrow with two heads (cf. $\twoheadrightarrow$ in Fig. 1, right)[4]. We are aware that infinite unfolding may come from more complex cycles in a graph-based metamodel. For example in Figure 1, we may consider the cycle between State and Transition as an infinite unfolding. Common textual and graphical metamodeling notations cannot easily characterize a cycle. Nevertheless, a reflexive relation may be derived from an OCL expression characterizing the cycle. For example, the reflexive reference nextStates on State in Figure 1 is derived as specified by the following OCL expression. This derived reference characterizes the cycle between State and Transition.

```
context State :: nextStates : Set( State ) derive :
        self .outgoing->collect( target );
```

According to this new notation, the classical example proposed in the left of Figure 1 is modified as shown in the right of Figure 1 in order to locally consider infinite structures (in our case to consider a possible infinite execution of a state machine). Note that several collections and cycles voluntarily keep the initial semantics based on a finite part of the model. For example, the collection of states (resp. transitions) which compose a state machine is unbounded but remains finite, and the transitive closure of

[4] Finite unfoldings where an explicit bound is known could be interesting. We could then add an annotation on relations, belonging to the same type as the upper bound of the multiplicity of collections. This extension is not taken into account in the scope of this paper.

the reflexive relation cause always terminate. So this syntax allows to clearly define in a metamodel the parts of a model which should be interpreted as infinite.

3.2 Formal presentation

We propose a formal definition relying on the previous intuitive presentation of our extended metamodeling facilities.

In the following formal definition, we assume that we have finite sets of meta-elements (`MetaElements`) and relations (`Relations`), i.e., a finite metamodel. We are also using `Elements` as the set of possible model elements without any type information.

Definition 1 (Infinite Model). *Let $\mathscr{ME} \subseteq$ `MetaElements` be a bounded set of meta-elements. Let $\mathscr{R} \subseteq \{\langle me_1, r, me_2\rangle \mid me_1, me_2 \in \mathscr{ME}, r \in$ `Relations`$\}$ be the bounded set of relations among meta-elements such that $\forall me_1 \in \mathscr{ME}, \forall r \in$ `Relations`, $card\{me_2 \mid \langle me_1, r, me_2\rangle \in \mathscr{R}\} \leq 1$.*

We define an infinite model $\langle E, L\rangle \in$ `Model`$(\mathscr{ME}, \mathscr{R})$ as a multigraph built over an unbounded set E of typed elements and an unbounded set L of labeled links such that:

$$E \subseteq \{\langle e, me\rangle \mid e \in \texttt{Elements}, me \in \mathscr{ME}\}$$
$$L \subseteq \left\{ \langle\langle e_1, me_1\rangle, r, \langle e_2, me_2\rangle\rangle \middle| \langle e_1, me_1\rangle, \langle e_2, me_2\rangle \in E, \langle me_1, r, me_2\rangle \in \mathscr{R} \right\}$$

We define the auxiliary type $Natural^{\omega} = \mathbb{N} \cup \{*, \omega\}$. $Natural^{\omega}$ is an extension of the `UnlimitedNatural` type provided by the OMG MOF specification. It is used to represent a range of possible numbers of instances. Unbounded finite ranges can be modeled using the $*$ value whereas unbounded infinite ranges can be modeled using the ω value. The type $Natural^{\omega}$ also comes equipped with the following order: $m < * < \omega$, for all $m \in \mathbb{N}$.

We aim at characterizing the presence of infinity both at the level of collections (i.e. the width of the underlying graph) and reflexive relations unfolding (i.e. the depth of the underlying graph).

First, regarding the width of the graph, we define the *upper* property which aims at distinguishing finite collections from infinite ones. Either for attributes or references (i.e., relation), a maximum number of instances of a target concept can be defined using the *upper* attribute, which value n is reflected in the following definition. Whether the upper bound of a relation is finite or infinite impacts the semantics (and implementation as well) of the model elements fetching operator *get* (cf. Section 4.2).

Definition 2 (Upper). *The upper property characterizes an upper bound n of a multiplicity of a given relation, this bound been taken from $Natural^{\omega}$.*

$$upper(\langle me_1, r, me_2\rangle \in \mathscr{R}, n \in Natural^{\omega}) \triangleq \langle E, L\rangle \mapsto$$
$$\forall \langle e, me\rangle \in E, me = me_1 \Rightarrow card(\{m_2 \in E \mid \langle\langle e, me_1\rangle, r, m_2\rangle \in L\}) \leq n$$

where the card function returns either $m \in \mathbb{N}$ or ω.

Second, regarding the depth of the graph, we introduce the property *ru_unstable* applying on primitive as well as derived relations. It requires the definitions of (maximal) model paths as the results of relation unfolding.

Definition 3 (Model Path). *Let $\langle E, L\rangle \in \texttt{Model}(\mathscr{ME}, \mathscr{R})$ be an infinite model. A model path is a relation path through model elements, i.e. a sequence of triples $\{\langle\langle e_i, me_i\rangle, r_i, \langle e_{i+1}, me_{i+1}\rangle\rangle\}_{i\in I} \in L$, where either $I = [0, sup[$ is finite ($sup \in \mathbb{N}$) or $I = \mathbb{N}$ is infinite. Relying on the previous model path definition, we also assimilate a model path to the (behavioral) trace of the model element creations.*

Definition 4 (Maximal Path). *Let $\langle E, L\rangle \in \texttt{Model}(\mathscr{ME}, \mathscr{R})$ be an infinite model. A maximal path is a model path, such that if it is finite, then the final element of the sequence has no relation to any element of the model.*

The property *ru_unstable* states that a given relation only gives rise to finite unfoldings, whatever the maximal model path considered. We need to focus on maximal paths as we express properties about possibly infinite unfoldings. Whether a given relation of a model satisfies the *ru_unstable* property or not impacts the semantics and implementation of the OCL *closure* and other iteration operators applied to this relation (cf. Section 4.3).

Definition 5 (Unstable Reflexive Unfolding). *Considering model paths as creation traces (cf. definition 3), a relation has only finite unfoldings if and only if it is unstable in any maximal model path π. This condition is rephrased as the following Linear Temporal Logic (LTL) property: $\Box\Diamond\neg r$ or equivalently $\neg\Diamond\Box r$. Relying on our model definition, it amounts to directly defining the following property:*

$$ru\_unstable(\langle me, r, me\rangle \in \mathscr{R}) \triangleq \langle E, L\rangle \mapsto \forall \pi \in maximal\,paths(\langle E, L\rangle)$$
$$I_\pi = \mathbb{N} \Rightarrow \exists i \in I_\pi, me_i \neq me \vee r_i \neq r$$

where I_π is the set of indexes of π and satisfies definition 3.

4 A Coinductive Semantics to Iterate Infinite Models

We discuss in this section the ways to manage model elements in the context of an infinite model (Subsection 4.1) and we propose a formal definition of the operators needed to querying such models. First we formalize the common operators for getting model elements (Subsection 4.2), and then we rely on them to formalize an alternative of the main iterators inspired from the OCL language (Subsection 4.3). We finally put into practice the proposed operators, among others for the manipulation of infinite state machines as defined in the right of Figure 1 (Subsection 4.4).

4.1 Reasoning on model elements: from finite to infinite model

Standard inductive semantics aims at defining finite data-structures as well as reasoning and programming with them. Therefore, an inductive structure comes naturally equipped first with an induction principle, allowing proofs by induction on the elements of this data-structure ; and second with a generic reduce programming primitive, allowing terminating recursive traversal of these data-structures. Induction principles are commonplace for reasoning about terminating algorithms and finite data-structures. It

comes in many flavours such as induction on natural numbers, lists, binary trees, etc. As for the reduce and alike operators, they are also pervasive in programming paradigms, mostly in functional languages, but also in Java (e.g. the `Iterator` interface) and OCL (the `iterate` construction that operates on a finite collection).

Dually, coinductive semantics aims at defining potentially infinite data-structure. Therefore, a coinductive data-structure comes equipped with a coinduction principle and also a produce operator. The coinduction principle, among various usages, is at work when typing compilation units in languages supporting separate compilation. In Java for instance, you can type-check a bunch of classes, even if they are totally abstract and don't contain any piece of code. In this respect, checking type safety of two mutually dependent classes `A` and `B` works as if you were producing an infinite proof under the form: `A` is type-safe if `B` is type-safe if `A` is type safe, etc. There, type-safety is only proved to be a stable relation, with no base case at all. On the contrary, type-checking a concrete method amounts to reason inductively on the code structure, with assumed well-typed parameters and local object creations as base cases.

The produce operator (dual of the reduce operator) aims at producing potentially infinite data (as it cannot obviously perform a terminating recursive full traversal of an infinite structure), through the repeated execution of a piece of code that generates new values each time, appended to the growing structure. The resulting structure is the limit of this maybe infinite creation process, much akin to the fractal structures resulting from infinite iterations of a subdivision process. Often enough, as it is the case for instance in stream processing languages, the output structure is produced by a piece of code that consumes/explores in turn another corecursive input structure, one element at a time.

These coinductive concepts are mandatory in order to define the formal semantics of our MOF extensions (as used in the right part of Figure 1) independently of a particular implementation. The infinite state sequences of a state machine (cf. right lower part) should follow the semantics of the metamodel, just as plain finite models follow the inductive semantics of the metamodel. These state sequences may be defined as being produced from infinite sequences of transition-enabling events, following the execution semantics of the state machine. A standard inductive viewpoint on these sequences would be for instance to define an `allInstances()` OCL constraint that checks that each $n+1^{th}$ state is the result of executing a transition from a n^{th} state. As each OCL operator is supposed to work on a collection of states taken as a whole, evaluating the constraint on an infinite model would yield a non-terminating behavior and no outcome at all. Moreover, how such a collection may be produced still remains an open question in this case.

As can be seen, coinductive semantics, which amounts to producing infinite proofs and data, may be found in various areas, even though not always presented as such. Defining and formalizing a (coinductive) structure of models will help at elaborating important and practical tools for infinite model manipulation. A coinductive semantics may be implemented in several ways: in a programming language with lazy constructs that will evaluate only the finite browsed part of the model, or with data stream primitives randomly producing new model elements consumed afterwards by the execution semantics when possible, or else with a prefetch semantics that estimates how many model elements should be evaluated in advance, even if not needed. Each such imple-

mentation is interesting in its own right as it corresponds to a well-known class of applications. In the remainder of this section, we propose a formalization of a coinductive model semantics and the implementation standpoint is discussed in the next section.

4.2 Getting model elements

On a model $m = \langle E, L \rangle$, we first assume the operator getRoots() which corresponds to a minimal set of model elements from which any other model element can be accessed (i.e., a covering set). The accessibility predicate is defined as the existence of a finite model path from a model element to another. A set of model elements is defined as minimal when it is a covering set such that no proper subset is also covering.

$$\begin{aligned}
&\mathit{accessibility}(e, e' \in E) \triangleq \exists \{\langle e_i, r_i, e_{i+1} \rangle\}_{i \in I} \in \mathit{model\ path}(\langle E, L \rangle), \\
&\qquad\qquad e_0 = e \wedge \exists i \in I, e_i = e' \\
&\mathit{covering}(S \subseteq E) \quad \triangleq \forall e' \in E, \exists e \in S, \mathit{accessibility}(e, e') \\
&\mathit{minimality}(S \subseteq E) \quad \triangleq \mathit{covering}(S) \wedge \forall e, e' \in S, \neg \mathit{accessibility}(e, e')
\end{aligned}$$

We can now specify the getRoots() operator which corresponds to the entry point facility on the model:

$$m.\mathsf{getRoots}() \in \{S \subseteq E \mid \mathit{minimality}(S)\}$$

We assume that the getRoots() operator return a finite set of roots. We note also that an alternative model definition based upon rooted multigraph may be adopted and would yield a non under-specified definition. In this case, the set of roots is uniquely defined.

Relying on the entry point previously defined by the getRoots() operator, we mainly consider the get() operator for getting model elements from a model $\langle E, L \rangle$. Usually, this operator allows to access to a property r from a model element e (written $e.r$ in OCL).

$$e.\mathsf{get}(r \in \texttt{Relations}) \triangleq \{e' \in E \mid \langle e, r, e' \rangle \in L\}$$

where $e \in E$. In our formalization, we assume that the return value of the get() operator is always a collection of model elements, tagged as either finite or infinite in our MOF extension and processed accordingly with the appropriate iterator. We are aware that in most model management APIs (e.g., EMF) or in the OCL query language, the return value may exhibit different types according to the known multiplicity of the relation.

4.3 Iterating model elements

We propose in this section a formalization of an alternative version – called coiterate – of the main generic OCL iterator iterate [8, § 7.6.5], from which all other iterators may be defined. Both iterators allow to browse a collection returned by the get() operator and to unfold a reflexive relation, for instance in order to compute its closure.

A particularity of OCL finite collections is that they are implicitly ordered. Indeed, the n^{th} element may easily be retrieved with the help of the iterate operator. It may merely appear as a design clumsiness in the API for collections in the finite case. Yet,

the ordering of elements is mandatory in the infinite case, as elements will be processed sequentially, one by one, and the user may observe intermediate results of this infinite computation. So we assume that collections, whether finite or infinite, are ordered.

The iterate operator processes the successive values of an (ordered) finite collection in order to compute the final value of an accumulator of any type. Dually to this semantics, the coiterate operator starts from an initial value which processing produces a potentially infinite collection of new values. This approach, which applies the coinduction principles, allows to produce a new collection either from an already existing infinite collection or more generally as the sequence of values built from successive assignments of a variable of any type.

In order not to depart too much from the iterate operator, from a syntactic viewpoint, we define coiterate as follows, first recalling the definition of iterate. We require that *coll*:Collection(A) possesses the three basic operations provided on Collection by OCL: isEmpty() which tests a collection for emptiness, first() which returns the first element of a collection, and append(elem:A) which appends a new element elem at the end of the collection. For the sake of readability, we also define the operation tail() which returns the collection without the first element[5]. Note that, as e_1, e_2 and e_3 are expressions and not values in the following definitions, we must use substitutions[6] in order to define recursively (resp. corecursively) these iterators.

$$
\begin{array}{l}
\mathit{coll}\text{->iterate(elem: } A\text{; acc : } B = e_1 \mid \text{acc = } e_2) \triangleq \\
\quad \mathit{if}\ \mathit{coll}\text{->isEmpty() } \mathit{then}\ e_1\ \mathit{else}\ \mathit{let}\ e_1' = e_2[\mathit{coll}\text{->first()} \mid \text{elem}][e_1 \mid \text{acc}]\ \mathit{in} \\
\quad \mathit{coll}\text{->tail()->iterate(elem: } A\text{; acc : } B = e_1' \mid \text{acc = } e_2)
\end{array}
$$

$$
\begin{array}{l}
\mathit{coll}\text{->coiterate(acc : } B = e_1 \mid \text{acc = } e_2\text{; elem: } A = e_3) \triangleq \\
\quad \mathit{if}\ e_1 = \text{null}\ \mathit{then}\ \mathit{coll}\ \mathit{else}\ \mathit{let}\ e_1' = e_2[e_1 \mid \text{acc}]\ \mathit{in} \\
\quad \mathit{coll}\text{->append(}e_3[e_1 \mid \text{acc}]\text{)->coiterate(acc : } B = e_1' \mid \text{acc = } e_2\text{; elem: } A = e_3)
\end{array}
$$

The coiterate operator starts from a finite collection *coll*, to which it will append a potentially infinite sequence of elements. For that purpose, it considers a variable *acc*, initialized with value e_1. From the current value of *acc*, if not equal to null, a new element *elem* with value e_3 built from *acc* must be appended to the current resulting collection, and the next value of *acc* is given by e_2. This process is repeated as long as *acc* is not null. So the resulting collection is finite if and only if *acc* finally becomes null, otherwise the collection is infinite. Moreover, both iterators are able to handle several accumulating variables of any name at once, provided the variable *elem* is defined.

4.4 Putting the coiterate iterator into practice

We illustrate how the coiterate iterator may be used, through the following two basic examples. In the first one, the infinite collection of the natural numbers is built and in the second one, the infinite collection of the squares of even numbers is built from the first collection.

[5] The operation tail() is defined in OCL by: *coll*->excluding(coll->first()).

[6] $e[u \mid u']$ is the expression e where any occurrence of sub-term u' has been replaced by u.

1. *Naturals* $\triangleq$

```
Set{}->coiterate(acc:Integer = 0 |
        acc = acc+1; elem = acc)
```

2. *Squares* $\triangleq$

```
Set{}->coiterate(acc: Collection(Integer) = Naturals |
        acc = acc->tail()->tail(); elem = acc->first()*acc->first())
```

Similarly, the operator coiterate may be used to browse the infinite collection of events to process (cf. eventToProcess in the right part of Fig. 1) and then to specify the simulation of a state machine. For instance, the following listing define the body of an operation simulate() on StateMachine which creates the trace (*i.e.,* a sequence of states) according to the events to process[7].

```
context StateMachine::simulate() : Sequence(State) body :
        Set{}->coiterate(
                acc : Sequence(EventOccurence) = eventToProcess ;
                current : State = self.initial |
                current = self.step(current, acc->first()) ;
                acc = acc->tail() ;
                elem = current
        )
```

Our definition of the iterate (resp. coiterate) iterator was shown to browse a collection (resp. create a collection). However, it seems to be an elegant way to use these operators in order to also (co)iterate over unfoldings of a reflexive relation. In the following listing, the coclosure operator of a reflexive relation is defined in terms of coiterate and corresponds to a breadth-first traversal of the underlying graph. From this generic coclosure operator we also specify an operation on *State* computing the potentially infinite set of reachable states from a given state:

```
context T::coclosure(relation) : Sequence(T) body :
        Set{}->coiterate(
                acc : Sequence(State) = Sequence{self} |
                acc = acc->tail()->union(acc->first(). relation
                        ->asSequence())
                elem = acc->first() ;
        )

context State :: reachableStates() : Sequence(State) derive :
        { self }->coclosure( nextStates );
```

[7] We consider the operation step() which returns the current state according to the event given as a parameter.

5 On the Implementation of Coinductive Operators

We have proposed in the previous section a formal specification to define and evaluate infinite models relying on coinduction principles. Stemming from this first milestone, our immediate next goal will be to explore pragmatical solutions to implement such principles.

In all cases, and by definition, it is not possible to store the entire infinite model in memory. When a causal dependency exists between the model producer and the model consumer, the model can be lazily interpreted (from the producer) in order to build, on demand (by the consumer), only the necessary model elements. In our example, the state machine may be lazily executed during a step-by-step simulation. Such lazily interpreted constructs have been recently proposed at the metamodel level in ATL [3]. In this case, infinite model processing is close to existing lazy interpretation (also called *call-by-need*) of a data structure [9] and relies on the following properties :

- the model elements in the infinite model are only built when it is necessary,
- once an infinite model element has been built, it is never built twice.

However, in some situations it may be necessary to relax these properties. For example, the last property implies *memoization*[8] and may be relaxed in some cases because it may render certain things impossible (e.g., because more elements may have to be remembered than system memory permits) or very inefficient (e.g., because of synchronization issues in concurrent systems). A non-memoized lazy evaluation also called *call-by-name* [10] may then be of interest. For example, such strategies have been explored in [4] as part of an approach for model persistence based on NoSQL databases.

Even if lazy interpretation seems promising, specifying the level of laziness is also important. In the context of the MDE, different solutions are possible. For instance, does model element creation entails creation of its references atomically? Or are these references also created on demand?

When the model consumption is disconnected from the model production, the order in which model elements arrive is out of control. Such an infinite model relies on the following properties (inherited from data stream [11]):

- model elements in the infinite model arrive online,
- once an element from an infinite model has been processed it is discarded or archived.

Different semantics may be considered depending on several options. For instance, model elements delivery must take into account the following options from [12]: *Pull vs. Push*, *Aperiodic vs Periodic* and *Unicast Vs. 1-to-N*. Then, mechanisms for model elements delivery may be inspired from known protocols in data stream, such as *Request/Response* (Aperiodic Pull), *polling* (Periodic Pull), *Publish / Subscribe* (Aperiodic Push) or *Broadcast Disks* (Periodic Push). Moreover, an info gatherer may be used to change the appearance of the real model elements delivery at the query language level. In this case, we have also to take into account the conformity points when model interpretation is coherent, for instance using an operation-based model representation [13].

[8] With memoization, computation of a model element is not repeated: the element is kept in a cache after its first usage.

6 Conclusion and Perspectives

The contributions of this paper are motivated by the need to define and manipulate infinite models (i.e. models whose comprehensive set of model elements is too large to be loaded or even not available). After pinpointing how current metamodeling formalisms prevent such situations, we first propose MOF extensions to locally characterize in a metamodel the infinite parts of the conforming models. Then we introduce a formal alternative semantics of the OCL operator iterate, called coiterate, providing an implementation-independent semantics for manipulating infinite models, and supporting specification and verification of complex situations involving infinite models. The coiterate operator can be used both to browse an infinite collection, and to compute the infinite closure of a transitive relation (or more generally a cycle in a model).

Such a coiterate operator can support the formal verification of operations manipulating infinite models. This verification activity can be partially automated by proof assistant which supports coinductive semantics. Among others, Coq is a valuable candidate in that respect [14]. This paper also discusses various possible implementations of the coiterate operator, whether for reasons of partial availability or partial loading of models.

More generally, iterate and coiterate iterators may be used for model transformation. Indeed, the accumulator may be a model (i.e., *modeling in the large*) which is finitely extended and returned (iterate) or indefinitely browsed in order to produce a new model (coiterate). If we abstract away the pieces of code used as arguments of these iterators, it turns out that iterate needs a function of type[9] $1+A\times B\rightarrow B$ whereas coiterate needs a function of type $B\rightarrow 1+A\times B$. In order to apply these iterators to model production or browsing, the types A and B may be generalized to dependent types that denote arbitrary predicates over a (mega-)model structure, aiming at identifying patterns of interest and providing entry points in these patterns. In this context, we would need functions with respective types $\forall e\in E.1+A(e)\times B(e)\rightarrow B(e)$ and $\forall e\in E.B(e)\rightarrow 1+A(e)\times B(e)$, where e are elements of the (mega-)model. Here, $A(e)$ and $B(e)$ represent query or creation patterns, the functions induce transformation rules and the iterators represent the execution of a global transformation engine. Finally, specifying models and model transformations with type predicates allows to talk about their respective properties within a single language of types. Turning questions about models into typing problems also brings a rich amount of results in the scope, about type checking, type inference and subtyping issues for models and model transformations at once, as investigated in [15].

To conclude with it, some substantial amount of work will be necessary to draw all the consequences of this promising approach for model transformation and to thoroughly compare and cross-fertilize it with existing solutions.

References

1. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley (2008)

[9] In abstract set algebra settings, "1" denotes the set with only one element, "+" denotes the disjoint sum and "×" the cartesian product.

2. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software and Systems Modeling (SoSyM) **11** (2012) 99–109
3. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: 14th ACM IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS). Volume 6981 of LNCS., Springer (2011) 32–46
4. Espinazo-Pagán, J., Cuadrado, J.S., Molina, J.G.: Morsa: A scalable approach for persisting and accessing large models. In: 14th ACM IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS). Volume 6981 of LNCS. Springer (2011) 77–92
5. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42** (2009) 22–27
6. Object Management Group, Inc.: Meta Object Facility (MOF) 2.4.1 Core Specification. (August 2011) Final Adopted Specification.
7. Object Management Group, Inc.: Unified Modeling Language (UML) 2.4.1 Infrastructure. (August 2011) Final Adopted Specification.
8. Object Management Group, Inc.: Object Constraint Language (OCL) 2.3.1 Specification. (January 2012)
9. Henderson, P., James H. Morris, J.: A Lazy Evaluator. In: 3rd ACM Symposium on Principles on Programming Languages (POPL), ACM (1976) 95–103
10. Douence, R., Fradet, P.: A systematic study of functional language implementations. ACM Transactions on Programming Languages and Systems **20**(2) (1998) 344–387
11. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: 21st ACM Symposium on Principles of database systems (PODS). (2002) 1–16
12. Franklin, M., Zdonik, S.: A framework for scalable dissemination-based systems. SIGPLAN Not. **32**(10) (1997) 94–105
13. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: 30th International Conference on Software Engineering (ICSE), ACM (2008) 511–520
14. Bertot, Y.: Coinduction in coq. CoRR **abs/cs/0603119** (2006)
15. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling (SoSyM) **6**(4) (2007) 401–414