Streaming model transformations: Scenarios, challenges and initial solutions

Jesús Sánchez Cuadrado[1] and Juan de Lara[1]

Universidad Autónoma de Madrid (Spain)
{Jesus.Sanchez.Cuadrado, Juan.deLara}@uam.es

Abstract. Several styles of model transformations are well-known and widely used, such as batch, live, incremental and lazy transformations. While they permit tackling advanced scenarios, some applications deal with models that are only available as a possibly infinite stream of elements. Hence, in *streaming transformations*, source model elements are continuously produced by some process, or very large models are fragmented and fed into the transformation engine. This poses a series of issues that cannot be tackled using current transformation engines. In this paper we motivate the applicability of this kind of transformations, explore the elements involved, and review several strategies to deal with them. We also propose a concrete approach, built on top of the Eclectic transformation tool.

Keywords: Model transformations, Streaming transformations, Transformation engines, Scalability

1 Introduction

Model-Driven Engineering (MDE) is increasingly being used to tackle problems of raising complexity, in scenarios for which current model transformation technology was not originally conceived [6, 27]. One such scenario is transforming models that are only available as a stream of model elements. While data stream processing has been investigated in the databases [1, 20] and XML [15] technical spaces, its application to MDE has been little investigated so far [10].

A streaming model transformation is special kind of transformation in which the whole input model is not completely available at the beginning of the transformation, but it is continously generated. Hence, it must be processed incrementally, as elements arrive to the transformation process. For instance, if we aim at processing tweets from Twitter, we can see tweets, users, hashtags, etc, as model elements that are processed as they are generated by the Twitter users. This model is indeed potentially infinite, and cannot be queried, matched or transformed at once. Nevertheless, a streaming transformation is not only useful for those cases in which the input model is inherently streamed and infinite, but it is also a way to deal with large models by feeding a transformation process incrementally, for instance to distribute a transformation, pipeline a transformation chain, or to avoid overflowing the memory of a machine.

In this paper we report our findings on the elements and challenges involved in streaming model transformations. We have looked into which features make streaming transformations different from other types of transformations, and we have identifed several challenges that must be tackled. Then we have explored several strategies that can be used to deal with such challenges, and we have implemented a concrete proposal into the Eclectic transformation tool [12]. The paper is motivated and illustrated by means of a selected example that showcases most of the elements of streaming transformations.

Organization. In Section 2, we analyse applicability scenarios for streaming transformations. Section 3 introduces a running example, identifying challenges to be tackled by streaming transformation engines. Section 4 deals with model element streams, Section 5 with transformation scheduling, and Section 6 with arbitrarily large models and collections. Section 7 evaluates the proposal. Section 8 reviews related research and Section 9 concludes.

2 Motivating scenarios

The problems involved in data stream processing have been investigated in the context of databases [1, 20], XML [15] and the semantic web [2, 24], where the main applications are directed to querying, filtering and aggregating streamed (sometimes unstructured) data. In contrast, model transformation techniques unfold their potential when applied to scenarios in which there is a transformation problem involved, either to convert already structured data or to give a model structure to unstructured data.

This difference in the applicability field implies that there is currently a lack of concrete examples and usage scenarios for streaming model transformation, which are needed to assess the potential of this new technique. For this reason we begin by introducing some possible scenarios and concrete examples.

Processing natural streams. Some systems naturally generate data continously, which might need to be transformed, e.g., for analysis or visualization. We distinguish two kinds of systems: (1) those which natively generate stream models (data conforming to a meta-model) and (2) when the data does not conform to a meta-model, but must be first converted to a model-based representation.

An example of (1) is the monitoring of a running system by generating model-based traces of its execution. This will be used as our running example.

An example of (2) is applying streaming transformations to *semantic sensor web technologies* [26]. This may include transforming native sensor data (e.g., temperature, precipitation) to the RDF format relating elements by Linked Data URIs, then further manipulating it, for instance to add information coming from other sources (e.g., amount of cars in a road segment) and to transform it to some other formalism to perform predictions (e.g., traffic jams depending on the weather conditions for certain road segments). As suggested in [25], data from physical sensors can be enriched with data from *social sensors*, like tweets, taking advantage of their attached spatial and temporal information and some defined microsyntax (like #hashtags, @usernames, subtags, or relations from predefined vocabularies, like e.g., for weather or emergency conditions).

The usefulness of model transformations in this scenario is to facilitate the implementation of stream-based applications in which there are an explicit or implicit transformation task involved. The next scenarios apply the notion of streaming data to solve problems in the model transformation domain.

Dealing with large models. An scalable solution to transform large models, is to incrementally feed the transformation process with model fragments. As suggested by [19], instead of loading a large model in memory and then transform it all at once, the model is first split into meaningful parts, which are sent to a stream. The transformation process deals with the elements of the stream incrementally, releasing resources as parts of the source model are transformed. In some way, this imitates lazy transformations [27], but using a push approach.

As a concrete example, let us assume we are reverse engineering a Java model into a KDM model [22]. The Java model would be available in some model repository, and the abstract syntax model of each Java class could be streamed separately (using lazy loading techniques [14]) to the transformation engine. The engine would transform each class individually, discarding all source elements and trace links no longer needed for transforming other classes (e.g., once a Java expression has been transformed to KDM it can be discarded).

Distributed transformations. The idea of streaming transformation can be used as a foundation to build distributed transformations. This is especially important to integrate MDE services in the Cloud [6, 7] since a large transformation could use different physical resources depending on their availability. The underlying idea is to replicate the same transformation in several execution nodes. Load balancing techniques would then be used to stream disjoint parts of the input model to such nodes. A shared repository could be used to store trace links and the output models, although other advanced techniques of distributed systems needs to be studied to improve scalability. Although this scenario is not addressed in this paper, we believe that the techniques explained here are complementary for developing distributed model transformations in practice.

Pipelining transformations. This scenario exploits the possibility of starting a transformation (within a transformation chain) as soon as target elements are generated by the previous transformation, in a similar way as Unix pipes. This permits taking advantage of multi-core architectures, by scheduling the first transformation in one core, and the subsequent transformations in different cores.

As an example, consider a parser that generates a concrete syntax model of a Java system (i.e., a low-level Java model), which is then transformed into an abstract syntax model (e.g., references between type definitions and type uses are resolved), and then into KDM. Using streams, each transformation can begin as soon as the previous one has finished processing a single Java class.

3 Running example and challenges

Assume we are interested in the reverse engineering of sequence diagrams from the execution traces of a running object-oriented program. The example is an

adaptation of the one in [5], in which the actual transformation used to process the execution traces and create the sequence diagrams is done off-line, after having generated all the traces in a text file. In our case, the transformation is on-line, that is, the sequence diagram is built as the execution traces are generated, by means of a streaming transformation. This enables the run-time monitoring of the system, and dealing with non-terminating systems.

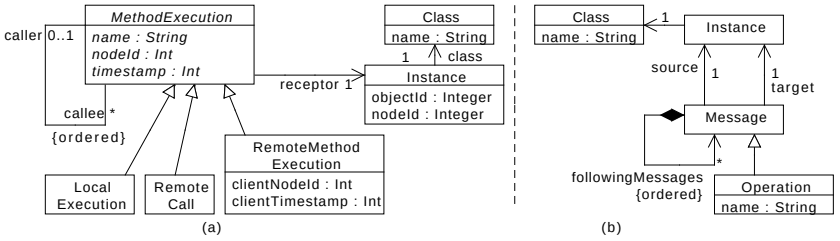

Fig. 1. (a) Trace meta-model (b) Sequence diagram meta-model (simplified)

The meta-models involved in the example are shown in Fig. 1. The *trace meta-model* represents the execution of methods (MethodExecution), including the information of which method performed the invocation, and the sequence of future method executions performed by itself (caller and callee references). Also, a method execution has a reference to the receptor instance. The meta-model is directed to distributed applications (e.g., Java RMI applications), hence there are three kinds of method executions: LocalExecution, normal method executions; RemoteCall, for invocations in the client side (e.g., over a proxy obtained using RMI); and RemoteExecution for the remote executions. Executions are identifed by the nodeId and timestamp attributes. A remote execution records the clientNodeId and clientTimestamp in order to identify the caller.

The *sequence diagram meta-model* represents messages from a source instance to a target instance (note that the source and target are explicitly represented by references, instead of by ids), and the sequence of messages that follows each Message (reference followingMsgs).

Our aim is to specify streaming transformations using regular constructs of rule-based model transformation languages. To illustrate the paper we have used the Eclectic transformation tool [12]. In particular, we use the mapping language which also allows attaching methods to metaclasses (helpers). The language can be seen as a simplified version of ATL [17]. Fig. 2 shows the corresponding transformation. Each LocalExecution is mapped to an Operation (lines 4-10). The source and target instances of the message are obtained from the local_context of the call (i.e., the object in which the call is performed, a helper in lines 29–31) and the receptor object. These bindings require rules that resolve the source instance to target instances, which is done in lines 20-22. Classes are also mapped (lines 24-26). Finally, the followingMsgs reference is filled by resolving those messages that correspond to the method executions calculated by the next_executions helper (lines 39–44), which basically retrieves all executions performed as part

```
1  mapping trace2seqdiagram(trc) -> (seq)
2  trc : 'platform:/resource/example/trc.stream'
3
4  from exec : trc!LocalExecution
5    to msg : seq!Operation
6
7    msg.source <- exec.local_context
8    msg.target <- exec.receptor
9    msg.followingMsgs <- exec.next_executions
10 end
11
12 from exec : trc!RemoteMethodExecution
13   to msg : seq!Operation
14
15   msg.source <- exec.remote_context
16   msg.target <- exec.receptor
17   msg.followingMsgs <- exec.next_executions
18 end
19
20 from src : trc!Instance to tgt : seq!Instance
21   tgt.class <- src.class
22 end
23
24 from src : trc!Class to seq!Class
25   tgt.class <- src.class
26 end
27
28 // Start of helper methods
29 def trc!MethodExecution.local_context
30   self.caller.receptor
31 end
32
33 def trc!MethodExecution.remote_context
34   self.caller.caller.receptor
35 end
36
37 // Find those executions that happen in the context
38 // of the current execution, but not before (excerpt)
39 def trc!MethodExecution.next_executions
40   trc!LocalExecution.allInstances.select { |me|
41     me.caller == self &&
42     me.timestamp > self.timestamp
43   }.union(...)
44 end
```

Fig. 2. Transforming traces to simplified sequence diagrams

of the execution of the current method (for simplicity only local executions are considered here). A RemoteMethodExecution is mapped similarly (lines 12–18), except that the source is obtained from remote_context which access the actual receptor object in the server side, and thus the client stub, which corresponds to the first caller, must be skipped.

3.1 Challenges

From the transformation engine point of view, this is a simple transformation, when applied in batch mode. However, it poses several challenges when the source model is processed in streaming. We next review these challenges, using the execution example shown in Fig. 3. The events are numbered in the order in which they are received by the streaming transformation.

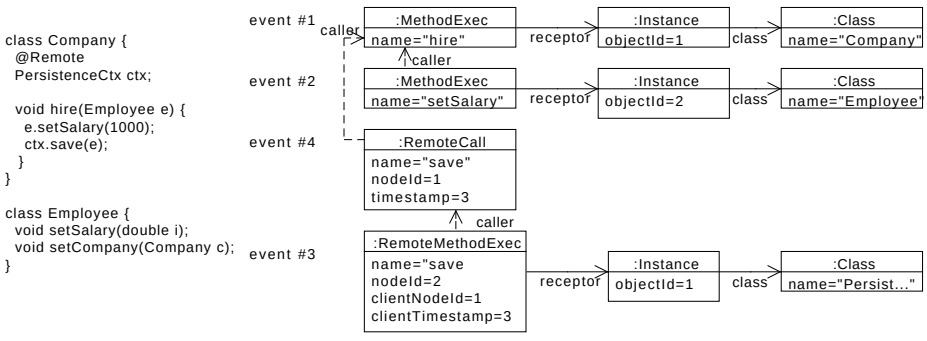

Fig. 3. Execution example.

- **Infinite model.** The input model is potentially infinite, as a program may be in execution indefinitely. The notion of infinite model has been studied in [10]. Similarly, the trace model that keeps the correspondences between source and target elements could also be infinite.
  In the example, each time a method is invoked over a local instance, MethodExecution, Instance and Class elements are created. They need to be transformed as they arrive from the stream, generating the corresponding trace links to allow bindings to be resolved (e.g., msg.target ← exec.receptor). As the program generating the execution traces may be in execution for a long time, strategies to reduce the amount of model elements and trace links kept are needed to avoid overflowing the memory of the machine.
- **Model element identity.** Transformation engines rely on the object identities, e.g. to compare two objects for equality. In our case, fragments of models can be streamed, and two or more fragments may contain the same element, but with different in-memory object identity.
  In the example, the processes generating the stream may create different Class elements to represent the same class in the program being analysed (i.e., in a distributed enviroment the same code is running in different machines). This implies that object identity may be lost. Additionally, in a distributed setting, a mechanism to serialize and deserialize fragments is needed.
- **Dealing with references.** A model fragment that is streamed may refer to other fragments that have already been streamed or that may be streamed in the future. Both cases are shown in the figure by the dashed arrows. Fragment event #2 refers to fragment event #1 through the caller reference (same for events #3 and #4). However, we do not want to emit all the elements of the referenced fragment again, but just to refer to a particular element. Hence, a mechanism to refer to elements in other fragments is needed.
- **Transformation scheduling.** In the example, obtaining the remote context (through expression self.caller.caller.receptor, line 34), may be a blocking operation since the caller may not be available when the rule is being processed (see reference from event #3 to #4). Some mechanism is needed to avoid stopping the execution of the whole transformation, and to resume the rule execution when the expected element arrives.
  In addition, rules must be executed as elements arrive, but the order is unknown. Thus, a flexible rule scheduling mechanism is needed.
- **Features with different semantics**. Some features normally available in model transformation languages are no longer adequate or their semantics has to be changed. An example is "all instances of", whose usual semantics is not valid in this context. This is so as all objects of a certain class cannot be generally available at a certain moment, either because they still need to arrive, or perhaps they have been discarded. Other features such as iterators on collections like select also need to be adapted, as proposed in [10].
  In the example, to obtain the executions that follows the current one (lines 40–43), the allInstances construct must be used. In both cases, a mechanism to process the elements as they appear are needed. In the case of allInstances an strategy to avoid dealing with a possibly infinite collection is also necessary.

As can be observed, streaming model transformations are an essentially different problem from other scenarios, such as live/change-driven [4] and incremental transformations [18, 16], in which the aim is to change the model (source model for in-place transformations, or target model for model-to-model transformations) as a response to changes in the source model. In our case the only change is the generation of new elements, but the source model can be infinite.

4 Specifiying model streams

A streaming transformation deals with model fragments that are continously made available. Hence, it is necessary to describe their characteristics so that the transformation engine can deal with them transparently.

In our approach, the streaming unit is the *model fragment*, made of one or more model elements which may have intra-fragment references (both containment and non-containment) and inter-fragment references (only non-containment, because the ultimate goal of them is to refer to an element not defined in this fragment).

Model fragments may need to be serialized if they are to be sent to the machine where the transformation is being executed. Thus, when creating and receiving a fragment, there are two main elements to take into account: model element identity and references. We have defined a small DSL to specify these features, among others. The stream description for the running example is shown in the following listing.

```
1  stream "dynamic_trace.ecore"
2  // Defining keys: simple, multiple, custom
3  key(Class) = name
4  key(Instance) = objectId, nodeId
5  key(MethodExecution) = { self.name + "_" +
6  self.nodeId + "_" + self.timestamp }
7  // inter-references
8  ref(MethodExecution.caller)
9  // Sliding windows
10 sliding for MethodExecution = 200 secs
11 sliding for Instance = 1000 elements
```

Model element identity. In the general case we cannot rely on plain object identity to compare model elements, as the elements of the stream may have been generated by a machine different from where the transformation is executed, as it is the case of the running example. This requires using the properties of the model elements to identify the objects (i.e., rely on value identity), similar to *keys* in the case of QVT-Relations [23]

Hence, we allow the key of an element to be specified in the stream description. Keys can be either simple, or composed of several attributes, or generated by an expression (lines 4, 5 and 6-7 respectively). Each time two elements of the same type are compared, the key value is used if a key has been specified. If the whole stream is generated in a single machine, the object identifier in this machine can be attached to each object prior to serialization.

Inter-fragment references. Our approach for inter-fragment references is based on creating a proxy per each referenced element. We do not rely on any particular technology, but we just create a new element (the proxy), of the same type as the referenced element, setting its key attributes (or attaching a "MemoryId"

annotation). The advantage is that, from the serialization point of view, inter-fragment references are not cross-references but just an annotation indicating that an element is a proxy, making it straightforward to implement and meta-modeling framework agnostic.

Upon arrival, our transformation engine replaces the proxy with the actual element if it was streamed before. To this end, the engine internally uses an associative table to keep the relationship between keys and actual elements. The case in which the actual element arrives after a proxy needs a special treatment, as discussed in Section 5.3.

In the DSL we allow specifying which references may hold a proxy (line 8). While this is not compulsory, we use this information to optimize the lookup and the replacement of proxies for the actual elements.

5 Transformation scheduling

Our approach to schedule streaming transformations builds on our previous work using continuations to schedule batch model transformations [11, 12], extended to consider the streaming setting, that is, rules fed incrementally by stream events and partial execution of navigation expressions. Our Eclectic transformation tool relies on an intermediate language, called IDC (Intermediate Dependency Code), to which high-level languages are compiled to.

IDC is a simple, low-level language composed of a few instructions, some of them specialized for model manipulation and transformation scheduling. IDC is compiled to the Java Virtual Machine (JVM). Fig. 4(a) shows an excerpt of its meta-model. Every instruction inherits from the Instruction abstract metaclass. Since most instructions produce a result, they also inherit from Variable (via InstructionWithResult) so that the produced result can be referenced as a variable.

The IDC language provides instructions to create closures, invoke methods, create model elements and set and get properties (Set and Get in Figure 4), among others. In IDC, there is no notion of rule, but the language provides a more general mechanism based on queues. Compilers for high-level languages are in charge of mapping actual rules to queues. A Queue holds objects of some type, typically source model elements and trace links. The ForAllIterator receives notifications of new elements in a queue, and executes the corresponding instructions. There are two special instructions to deal with queues: Emit puts a new object into a queue, while Match retrieves an element of a queue that satisfies a given predicate. If such an element is not readily available, the execution of this piece of code is suspended into a *continuation* [9] until another part of the transformation provides the required value via an Emit.

In the following we discuss, in the context of IDC, the elements involved to schedule a streaming model transformation.

5.1 Feeding transformation rules

Each time a new model fragment arrives, the source pattern of the transformation rules must be evaluated to trigger a rule if there is a match. Figure 4(b.1)

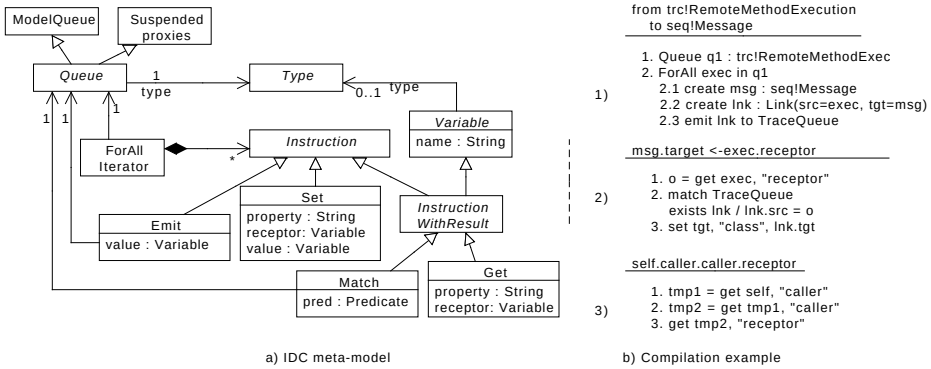

Fig. 4. (a) Excerpt of the IDC meta-model, (b) Compilation example between the Eclectic mapping language and IDC

shows how the rule to transform RemoteMethodExecutions is compiled to IDC. We create one queue per each type in the source pattern, and a ForAllIterator instruction which acts as a closure that is invoked each time a new element in the queue appears. In the example, a new Message element is created, as well as the corresponding trace link which is sent (via the emit instruction) to a default queue which is in charge of processing trace links (TraceQueue).

This mechanism permits the execution of rules on demand, as queues are filled. ForAllIterator instructions can be nested allowing complex patterns to be detected, and, as we will see in Section 6.1, our queues have "memory" (they have a sliding window), which is needed to allow the nesting of iterators.

In contrast to batch transformations, we needed to check that the rule has not been applied before for the current element, since an element with the same key may have arrived before. To this end we have an index with the received model elements, which is checked before feeding a queue. As explained in Section 3.1 this is the case with Class elements.

5.2 Resolving source-target relationships

A common operation in model-to-model transformations is to retrieve a target element from a source one already transformed by some rule. In the example this is achieved using a binding construct, such as msg.target ← exec.receptor.

We compile a binding as shown in Figure 4(b.2). (1) The expression to the right is compiled using regular model manipulation instructions, a Get in this case. Then, (2) the source element resulting from evaluating the expression, o, is used to match a trace link in the TraceQueue whose source is precisely o. If such trace already exists (i.e., it has been previously added with an Emit, as in Figure 4(b.1)), it is immediately retrieved. If not, the execution of the rule is stopped, and a request is placed in the queue so that the rule is resumed when some Emit instruction generates the trace link satisfying the request.

This approach has the advantage of its flexibility, since rules can be matched and applied in any order. In a streaming setting, rules can be matched and

executed as elements arrive: if a binding needs a source element that has not been processed yet, the rule will wait, letting other rules start their execution.

5.3 Evaluating expressions

When evaluating a navigation expression over a streamed model it may happen that part of the navigation path is not available yet. In our approach this can be detected because the result of getting a property is a proxy object. Thus, the evaluation of the expression must be suspended until the real object arrives. This may in turn suspend the rule that depends on the evaluation of the expression.

We use a similar approach as for resolving trace links, applied to change the semantics of the Get instruction to deal with incomplete models. It is worth noting that this design is transparent to the high-level language, which see property access as a regular Get, as illustrated in the compilation example of Figure 4(b.3).

The process is as follows. Given an instruction such as get self, "caller" we check whether the receptor object or the result of the instruction is a proxy, and we try to resolve the proxy with one of the already streamed elements. If not, the evaluation of the expression is suspended into a continuation, placing a request in a queue (*Suspended proxies*). Later, as new objects arrive they are passed to this queue, to check if some of them satisfies one or more of the enqueued requests, in order to resume the suspended Get instruction.

6 Infinite models

Streaming model transformations deal with possibly very large models, whose size is unknown. This requires strategies to reduce the memory footprint of the transformation process. Besides, the fact that the whole model is unknown from the beginning implies that some collection operations must be adapted to the new setting. In this section we present our approach to both issues.

6.1 Reducing memory footprint

Model transformation engines typically keep the source model, the target model and the traceability links in main memory. In many practical scenarios this is the best alternative, but when the source model is expected to be very large, alternative strategies to reduce the memory footprint are needed. So far, we have considered two approaches: sliding windows and using secondary storage.

Sliding windows. A direct mechanism to deal with an infinite data stream is to use a sliding window. In our setting, both source elements and trace links outside the window will be discarded. As noted in [1], this is an approximation mechanism that may produce an incomplete target model, although in some scenarios it is acceptable to assume this limitation.

In our approach sliding windows are specified with the DSL (see lines 10–11 in the example). There are two types: windows based on time (e.g., 200 seconds)

and on a number of elements of a given type (e.g., 1000 elements). A sliding window works in a "first-in, first-out fashion", so that the first element that arrived is the first element to be discarded when the window must be "moved". When a source element is discarded, any other data structure that refers to it must be discarded as well. In our case, they are the trace links, the continuations created with a Match that expects a trace link with such source element, and the index keeping the already streamed objects by key.

Please note that, when defining the windows, it is important to consider the expected amount of data for each type. In the example we decided never discard Class objects, as the number of classes in a system is limited.

Using secondary storage. If we want to guarantee that all bindings and proxies are resolved (provided the corresponding elements are eventually streamed), a solution would be to resort on a model repository, such as Morsa [14], to store all or part of them. The main problem is that accessing the repository may slowdown the transformation execution. Hence, this strategy may be practical depending on the pace of stream, and therefore it will be best suited for a distributed scenario in which load balancing is possible (see Section 2).

As an optimization we would like to use asynchronous I/O for accessing secondary storage. This approach fits naturally in our continuation-based scheduling algorithm, since the access to the repository can be scheduled in a different thread, storing the rule execution into a continuation, and so other elements in the stream can be processed. When the repository provides the result, the rule is seamlessly resumed.

6.2 Collection operations

The implementation of collection operations such as select, collect, or allInstances need to be adapted to take into account that the source model is not completely available from the beginning. In our setting, this problem can be seen as a simplification of the incremental evaluation of OCL expressions, in which there are only addition events (elements are not deleted).

There are several approaches proposed in the literature [18, 16, 4], but we have adapted and implemented the *active collection operations* proposed in [3] into our transformation engine. For space reasons we just outline some of its elements. Fig. 5 shows the API of our implementation.

We have added two extensions to the original ImmutableList type of IDC: ActiveGenerator and ActiveOperation. The former is a collection in which elements are initially injected from the stream. The ActiveAllInstances is connected to a model queue that provides elements of the corresponding types as they arrive (e.g., MethodExecution.allInstances), whereas ActiveGet is used to retrieve elements from a multiple-valued feature (e.g., self.callee).

The second extension reifies collection operations as classes (ActiveOperation and operation subclasses such as ActiveSelect and ActiveCollect), so that an operation is kept active as an object that receives events through a source. A source

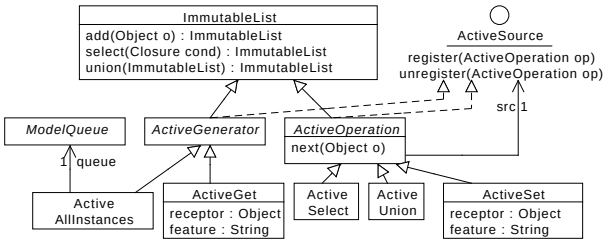

Fig. 5. Excerpt of the API of our active collection operations implementation.

is represented by the ActiveSource interface, which permits registering and deregistering an ActiveOperation. Given an expression such as the one in lines 39–44 in Fig. 2, a tree of active operations is constructed. When an element arrives, it is propagated from an active generator to the root.

Currently, we do not permit operations such as size or indexOf, as their semantics cannot be naturally aligned to a streaming setting. Finding out an appropriate semantics for these operations is left for future work.

7 First results and evaluation

We have implemented a proof of concept streaming model transformation engine on top of the Eclectic transformation tool[1] [12], using the techniques presented in the previous sections. To evaluate our approach we carried out three initial experiments [2], which stress different elements of our approach (corresponding to three scenarios explained in Section 2).

Natural streaming. We used the running example to test the first versions of our implementation. Then, we built a simulator to generate execution traces indefinitely, to feed the transformation. The mechanisms proposed in the paper allowed us to keep the simulator running for some time, using different sizes of sliding windows and available memory (from 24 MB to 256 GB, generating between 10.000 and 100.000 execution traces).

Dealing with large models. We injected into the Morsa repository [14] the models provided in the Grabats 2009 contest [3]. They represent Java projects (conforming to the Eclipse JDT meta-model) ranging from 70,000 to 500,000 elements (only injecting the largest model requires a setting with 3 GB RAM). To test the possibility of dealing with such large models, we implemented a transformation from JDT models to KDM. It transforms classes, methods, fields and resolve types, and therefore only parts of the source model needs to be in memory at

[1] Source code and examples are at `http://sanchezcuadrado.es/projects/eclectic`

[2] We have run the tests in an Intel i7 Quad Core, with 8 GB RAM, configuring the JVM with different heap sizes (up to 2GB).

[3] `http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs\_2009\_Case\_Study`

a given time. We used the load-on-demand facility of Morsa to incrementally feed the transformation, which allowed us to transform even the largest model (requiring 2 GB RAM, taking 16 minutes).

Pipelining transformations. We implemented a simple pipeline with two processes. The first process was in charge of parsing individual Java files into an AST (using the JDK's parser). The AST representing each class was then transformed into the MoDisco Java meta-model. In this case we have considered compilation units, classes and methods, and the inheritance reference between classes. We compared the execution time of performing the transformation in batch mode (parsing all models at once and then transforming) against scheduling the transformation two threads: parsing and transforming. Our streaming approach premits that, as soon as the parsing thread generates the AST of a file, it is passed to the transformation thread. We have tested with projects between 2,000 and 15,000 Java files (roughly 30,000 and 300,000 objects), and our results showed an speedup between 10% and 15% for the threaded approach. Even more, if we manually release resources not needed for subsequent executions (compilation units and method declarations in this case), speedup increases upto 10%, and memory footprint decreases 25%. As future work we aim at automatically identifying in which case resources can be safely released.

All in all, this initial evaluation shows the feasability of the approach, but more work is still required. For instance, this experience taught us that we had a few memory leaks which become very relevant in this setting, and that a mechanism to discard parts of the target model or to incrementally store it in a model repository is needed if the target model grows too large. Another future line of work is to evaluate how Event Stream Processing engines, such as Esper[4], could be used as a backend for the transformation engine.

8 Related work

Data stream processing has been investigated in the database community, proposing extensions for SQL and mechanisms for sliding windows, sampling and summarization [1]. Adapting query language designs and sliding windows implementation techniques is particularly interesting for our case [20, 21].

Works dealing with the processing of XML are also focussed on providing query facilities [15] or in the case of XLST, simple transformations (in-place substitution). Notably, STX is a variant of XSLT intended for streaming transformations of XML documents, based on SAX events instead of DOM [8]. These approaches could be used to complement our work, in the pattern matching phase, which we have currently implemented just by nesting *forall iterators*.

Proposals such as the *semantic sensor web technologies* [26] requires processing streamed semantic data, typically in the form of RDF triples, which can be queried with SPARQL extensions [13]. As noted in Section 2 our approach could be applicable to this context to data format transformations and to integrate data from heterogenous sources.

[4] `http://esper.codehaus.org/`

In [10] the authors provide a formal foundation for infinite models, as well as a redefinition of some OCL operators to tackle infinite collections using coalgebra. They identify transformations of such infinite models as a challenge. Lazy model transformations [27] somehow deal with the converse scenario we tackle here: on-demand generation of the target model. This scenario is useful if only some part of the generated model is needed, which is produced on-demand. That is, target elements are only produced when they are accessed. Change-driven transformations [4], incorporate the notion of *change* (in the source model) as a first-class concept in transformation languages. While this approach can be used to implement, e.g., incremental transformations, our approach enables the uniform specification of transformations, as if they were designed for a batch scenario, but are applicable for streaming data.

Techniques for incremental transformations are closely related [16, 18], but taking into account that in our case just additions need to be considered. Thus, we have used continuations to schedule the transformation execution [11, 12] and *active collection operations* [3] to implement infinite collections.

9 Conclusions and future work

In this paper we have presented our approach to streaming model transformations. We have motivated the problem by presenting four applicability scenarios, and providing a complete example. From the example we have derived the set of challenges that has driven our proposal, which includes mechanisms for specifying model fragments, transformation scheduling and dealing with infinite models. Our first experiments show promising results, not only to deal with natural streams, but also to deal with large models and to take advantage of multicore architectures. Additionally, we contribute a prototype implementation for the Eclectic transformation tool. To the best of our knowledge, this is the first model transformation engine with this capability.

As future work, we plan to perform further experiments, and to improve our implementation, for instance to allow the incremental store of the target model in a model repository and to take advantage of asynchronous I/O. Finally, we aim at using streaming transformations to implement distributed transformations.

Acknowledgements. This work was funded by the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139) and the R&D programme of the Madrid Region (project "e-Madrid" S2009/TIC-1650).

References

1. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16. ACM, 2002.
2. D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus. Querying rdf streams with c-sparql. *SIGMOD Record*, 39(1):20–26, 2010.
3. O. Beaudoux, A. Blouin, O. Barais, and J.-M. Jézéquel. Active operations on collections. In *MoDELS*, volume 6394 of *LNCS*, pages 91–105. Springer, 2010.

4. G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations - change (in) the rule to rule the change. *SoSyM*, 11(3):431–461, 2012.
5. L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE TSE*, 32(9):642 –663, 2006.
6. H. Brunelière, J. Cabot, and F. Jouault. Combining Model-Driven Engineering and Cloud Computing. In *MDA4ServiceCloud'10 workshop at ECMFA 2010)*, 2010.
7. M. T. Cauê Clasen, Marcos Didonet Del Fabro. Transforming very large models in the cloud: a research roadmap. In *Workshop on MDE on and for the Cloud*, 2012.
8. P. Cimprich. Streaming transformations for xml (stx) version 1.0 working draft. *Available on http://stx. sourceforge. net/documents/spec-stx-20040701. html*, 2004.
9. W. D. Clinger, A. Hartheimer, and E. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.
10. B. Combemale, X. Thirioux, and B. Baudry. Formally defining and iterating infinite models. In *MoDELS'12*, volume 7590 of *LNCS*, pages 119–133, 2012.
11. J. S. Cuadrado. Compiling ATL with Continuations. In *Proc. of 3rd Int. Workshop on Model Transformation with ATL*, pages 10–19. CEUR-WS, 2011.
12. J. S. Cuadrado. Towards a family of model transformation languages. In *ICMT'12*, volume 7307 of *LNCS*, pages 176–191. Springer, 2012.
13. E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Int. Sys.*, 24(6):83–89, 2009.
14. J. Espinazo-Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *MoDELS*, volume 6981 of *LNCS*, pages 77–92. Springer, 2011.
15. T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, Dec. 2004.
16. D. Hearnden, M. Lawley, and K. Raymond. Incremental model transformation for the evolution of model-driven systems. In *MoDELS'06*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.
17. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008.
18. F. Jouault and M. Tisi. Towards incremental execution of atl transformations. In *ICMT'10*, volume 6142 of *LNCS*, pages 123–137. Springer, 2010.
19. D. S. Kolovos, R. F. Paige, and F. Polack. The grand challenge of scalability for model driven engineering. In *MoDELS Workshops*, volume 5421 of *LNCS*, pages 48–53. Springer, 2008.
20. J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1), 2009.
21. Y.-N. Law, H. Wang, and C. Zaniolo. Relational languages and data models for continuous queries on sequences and data streams. *ACM Trans. Database Syst.*, 36(2):8, 2011.
22. KDM, v1.0. `http://omg.org/spec/KDM/1.0`.
23. OMG. QVT, v1.1, 2011. http://www.omg.org/spec/QVT/1.1/.
24. D. L. Phuoc, J. X. Parreira, and M. Hauswirth. Linked stream data processing. In *Reasoning Web*, volume 7487 of *LNCS*, pages 245–289. Springer, 2012.
25. T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *WWW*, pages 851–860. ACM, 2010.
26. A. P. Sheth, C. A. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, 2008.
27. M. Tisi, S. M. Perez, F. Jouault, and J. Cabot. Lazy execution of model-to-model transformations. In *MoDELS'11*, volume 6981 of *LNCS*, pages 32–46, 2011.