Streaming Model Transformations By Complex Event Processing

Conference Paper · September 2014

DOI: 10.1007/978-3-319-11653-2_5

3 authors:

Istvan David
Université de Montréal
28 PUBLICATIONS **246** CITATIONS
SEE PROFILE

István Ráth
Budapest University of Technology and Economics
70 PUBLICATIONS **1,575** CITATIONS
SEE PROFILE

Daniel Varro
Budapest University of Technology and Economics
223 PUBLICATIONS **6,017** CITATIONS
SEE PROFILE

Streaming Model Transformations By Complex Event Processing[*]

István Dávid[1], István Ráth[1] and Dániel Varró[1,2,3]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
Magyar tudósok krt. 2., 1117 Budapest, Hungary
`davidi@inf.mit.bme.hu, {rath, varro}@mit.bme.hu`
[2] DIRO, Université de Montréal, Canada
[3] MSDL, Dept. of Computer Science, McGill University, Montréal, Canada

Abstract. Streaming model transformations represent a novel class of transformations dealing with models whose elements are continuously produced or modified by a background process [1]. Executing streaming transformations requires efficient techniques to recognize the activated transformation rules on a potentially infinite input stream. Detecting a series of events triggered by compound structural changes is especially challenging for a high volume of rapid modifications, a characteristic of an emerging class of applications built on runtime models.
In this paper, we propose a novel approach for streaming model transformations by combining incremental model query techniques with complex event processing (CEP) and reactive (event-driven) transformations. The event stream is automatically populated from elementary model changes by the incremental query engine, and the CEP engine is used to identify complex event combinations, which are used to trigger the execution of transformation rules. We demonstrate our approach in the context of automated gesture recognition over live models populated by Kinect sensor data.

Keywords: streaming model transformations, complex event processing, live models, change-driven transformations

1 Introduction

Scalability of models, queries and transformations is a key challenge in model-driven engineering to handle complex industrial domains such as automotive, avionics, cyber-physical systems or ubiquitous computing. The maintenance and manipulation of large models identifies unique scenarios addressed by a novel class of model transformations (MT) to overcome the limitations or extend the capabilities of traditional (batch or incremental) MT approaches.

[*] This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) and MONDO (EU ICT-611125) projects partly during the third author's sabbatical.

Change-driven transformations [2] consume or produce changes of source and target models as their input or output models to enable transformations over partially materialized models and to reduce traceability information. Streaming transformations are defined [1] as a “special kind of transformation in which the whole input model is not completely available at the beginning of the transformation, but it is continuously generated.“ An additional class of streaming transformations aims to tackle huge models by feeding a transformation process incrementally (keeping only a part of the model in memory at any time).

In the current paper, we identify and address a novel class of streaming transformations for live models where the models themselves are not necessarily huge or infinite, but they change or evolve at a very fast rate (for instance, 25 times per second), and it is the stream of model changes that requires efficient processing. We propose a novel technique for streaming transformations to process these event streams in order to identify a complex series of events and then execute model transformations over them in a reactive way.

Our contribution includes a domain-specific event processing language for defining atomic events classes (from elementary or compound model changes using change patterns [2]) and combining these events into complex patterns of events. We also propose a general, model-based complex event processing architecture with a prototype engine Viatra-CEP to process rapidly evolving event streams. We also include an initial scalability assessment of the framework on a live model transformation scenario.

Our approach keeps the advantages of change-driven transformation as models can be partially materialized, since the processed event stream carries over only few relevant contextual model elements but not the models themselves. Instead, incremental model queries observe the model and publish relevant structural changes as atomic events in an event stream. Then this stream is processed by integrating known techniques from *complex event processing* (CEP) [3] to identify and handle a complex series of events.

In the rest of the paper, in Section 2, we introduce a case study of gesture recognition over live models used as a running example. The core ideas of our approach are presented in Section 3 while Section 4 presents an integrated tool set as a proof-of-concept. We carry out an initial performance of the approach in Section 5. Finally, related approaches and tools are described in Section 6 and Section 7 concludes our paper.

2 Case study: gesture recognition by live models

Our approach will be demonstrated on a gesture recognition case study. The use case is based on our preliminary work [4], presented earlier at EclipseCon Europe 2012, but without using the framework described in this paper.

In the case study, a human body is observed by optical sensors. The stream of data from the sensors (Microsoft Kinect [5] in our case) carries the spatial position of the hands, wrists, knees, etc. This stream is continuously processed and its data is stored in a *live model*, technically, an EMF model maintained via a Java based API [6]. Every time the optical sensors capture a new frame, the model is updated with the appropriate spatial data. The sensors process 25 frames per second, resulting in 25 model update transactions each second. The complexity of the scenario arises from the frequent changes the model undergoes. Executing model transformations on such a model poses several problems, since it would become obsolete quickly after being loaded into the memory. Moreover, model update transactions affect multiple model elements.

Figure 1 shows an excerpt from the domain metamodel [6], containing the head and the right arm. Similar metamodel elements describe the other three limbs of the body.

In this case study, we aim at recognizing a gesture in order to control a PowerPoint presentation with it. On the recognized gesture, the presentation advances to the next slide, therefore the gesture is referred to as the *forward gesture*. In our presentation [4] there is also a *backward gesture* to move back to the previous slide.

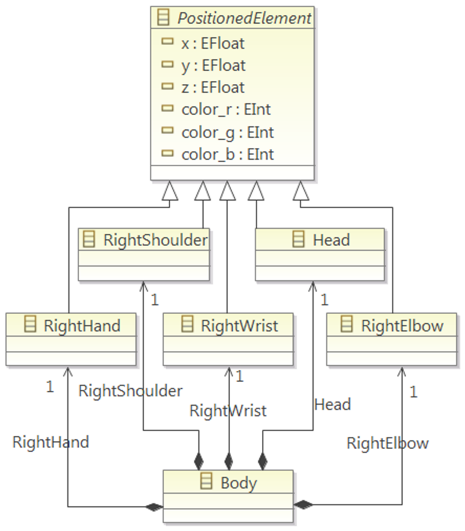


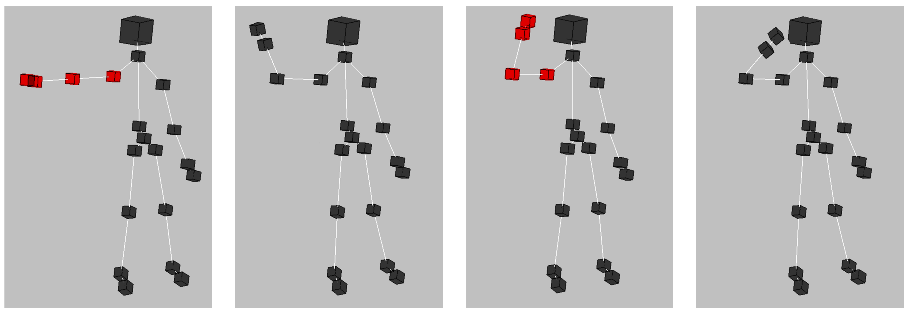
Fig. 1: Excerpt from the domain metamodel. [6]

As illustrated in Figure 2, the *forward gesture* consists of two postures: the *forward start* and the *forward end*. To recognize the gesture, the series of these two postures needs to be identified. Postures are considered as certain *states* of the body, which are described with a *range* or interval of spatial data. For example, the *forward start* posture is defined by the right arm being approximately stretched roughly to the height of the shoulder. Determining whether the arm is stretched is achieved by continuously measuring the angle between the upper and lower arm and smoothing the resulting stream of spatial data by a moving average transformation [7].

Processing a series of postures could be interpreted as a state machine, in which the states represent postures and transitions are triggered if a body leaves the valid range of the state and enters another. For instance, the body initiates the *forward start* posture by first entering the posture (*forward start found*), then leaving it (*forward start lost*) after a certain amount of time.

3 Overview of the approach

First, in Section 3.1, we provide a taxonomy (illustrated in Figure 3) on structural model changes and events (Section 3.1). In Section 3.2 we propose a novel approach for modeling and processing these changes as complex events in order



(a) Forward start found. (b) Forward start lost. (c) Forward end found. (d) Forward end lost.

Fig. 2: Body postures with the key context of the human body highlighted.

to support streaming transformations. In Section 3.3, the detection of complex event processing is briefly discussed.

3.1 A taxonomy of structural model changes

Elementary and compound structural model changes We define *elementary* changes as the most basic modifications applied on the model which cannot be refined into multiple modification steps. For example, in the case study in Section 2, such an elementary change would be moving the body's right hand on the x-axis, since it would require changing only one attribute of a `PositionedElement`. (See Figure 1.) Elementary model changes in this case are handled by the Eclipse Modeling Framework (EMF) [8] and its notifier/adapter techniques enabled by the EMF Notification API.

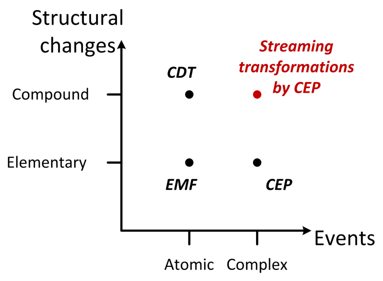


Fig. 3: Structural changes vs. events

On the other hand, *compound* changes consist of multiple elementary changes between two states (snapshots) of the model (called the pre-state and the post-state). For example, if the whole right arm is moved, the elbow, the wrist and the hand are moved consequently, i.e. the change affects multiple model elements. The techniques of change-driven transformations (CDT) [2] are capable of identifying compound structural changes by using *change patterns* [2,9,10]. Change patterns observe the delta between the pre-state and the post-state irrespective of how those states were reached, thus they abstract from the actual trajectories in the state space.

Atomic and complex events In our work, we consider both atomic and compound structural changes as *atomic events* in the event stream. This setup allows the use of events of different granularity. An atomic event is specified by its *type*, a set of model elements passed as *parameters* and a *timestamp*. *Complex events* are built up from sequences of atomic events and other complex events, using operators of an event algebra. Common operators enable the definition of events *following* other events, mutually *prohibited* events, or events *occurring within a given time window*.

Complex event processing (CEP) [3] techniques provide guidance on how to evaluate the stream of atomic events in order to detect complex events. Unfortunately, most CEP tools do not integrate well with existing model management frameworks (like EMF) and significant programming effort is required to translate elementary and compound structural model changes originating from a modeling tool into event types appropriate for a CEP engine.

In our work, we aim at combining the benefits of CDT and CEP resulting in a novel technique for identifying arbitrarily complex change events of compound structural changes.

3.2 Changes, events and streaming transformations

In this section, we demonstrate how streaming transformations can be defined by building upon well-established model query and transformation languages by elaborating the case study of Section 2. First, model queries will be used to identify the current state of the model and automatically publish notifications on relevant state changes in the form of atomic events. Then these atomic events will be combined into complex events using operators of an event algebra. Finally, we define transformation rules that are activated by a complex event.

Model queries for structural constraints

Model queries capture structural constraints of a model. In this paper, we employ the graph pattern language IQPL used by EMF-IncQuery [11]. This choice is motivated by the high expressiveness of the language and the incremental query evaluation strategy of EMF-IncQuery, which allows the sending of notifications upon the change of the result set of queries.

Listing 1 presents the graph pattern depicting the *Forward start* posture, as presented in Figure 2a. The pattern is parameterized with the spatial data of the right arm (consisting of the right hand, the right elbow and the right shoulder); the head; and the body the previous parts belong to. Accordingly, joins over the model are defined to describe this relationship in Lines 8-11. The *Forward start* posture requires a stretched right arm to be detected, but the arm shall not be held higher than head level (see Lines 13-14 and 16-17, respectively).

The latter one is a negative pattern call, which prohibits the occurrence of the `rightHandAboveHead` pattern presented in Listing 2. The pattern compares the spatial coordinates of the right hand and the head by their y coordinate. In Lines 8-9, the y coordinate of the right hand and the head is bound to the `RHy`

```
1  pattern ForwardStart(
2    B: Body,
3    RH: RightHand,
4    RE: RightElbow,
5    RS: RightShoulder,
6    H: Head)
7  {
8    Body.Head(B, H);
9    Body.RightHand(B, RH);
10   Body.RightElbow(B, RE);
11   Body.RightShoulder(B, RS);
12
13   find
14   stretchedRightArm(B, RH, RE, RS);
15
16   neg find
17   rightHandAboveHead(B, RH, H);
18
19 }
```

Listing 1: ForwardStart posture

```
1  pattern rightHandAboveHead(
2    B: Body,
3    RH: RightHand,
4    H: Head)
5  {
6    Body.RightHand(B, RH);
7    Body.Head(B, H);
8    RightHand.y(RH,RHy);
9    Head.y(H,Hy);
10
11   check(
12     MovingAverageCalculator::
13       getCalculator("HY").
14       addValue(Hy).movingAvg <
15     MovingAverageCalculator::
16       getCalculator("RHY")
17       .addValue(RHy).movingAvg
18   );
19 }
```

Listing 2: rightHandAboveHead

and `Hy` variables, respectively. The variables are evaluated in a *check* block in Lines 11-18 by invoking a Java based `MovingAverageCalculator` using Xbase syntax [12]. The details of the `rightHandAboveHead` pattern are omitted for space consideration.

Defining atomic events

In order to define atomic events, we propose an event processing language called the VIATRA-CEP Event Processing Language (VEPL). We built upon the result set of model queries to identify relevant structural changes, i.e. we identify when a new match is found for a model query or when an existing match is lost. These compound changes constitute the atomic events in our approach. Formally, an atomic event is specified as $a = (t, \mathcal{P}, d)$ where $a.t$ denotes the type, $a.\mathcal{P}$ is a list of parameters and $a.d$ is a timestamp of the event.

```
1  IQPatternEvent ForwardStartFound(B: Body)
2  {
3    iqPatternRef:
4      ForwardStart(B, _RH, _RE, _RS, _H)
5    iqChangeType:
6      NEW_MATCH_FOUND
7  }
8
9  IQPatternEvent ForwardStartLost(B: Body)
10 {
11   iqPatternRef:
12     ForwardStart(B, _RH, _RE, _RS, _H)
13   iqChangeType:
14     EXISTING_MATCH_LOST
15 }
```

Listing 3: Atomic event types

Listing 3 presents two atomic events reusing the graph pattern from Listing 1. Pattern `FSFound` describes the event when the *Forward start* posture is found (Figure 2a), while pattern `FSLost` describes the event when the *Forward start* posture is lost (Figure 2b).

Both atomic events are parameterized with a `Body` parameter (Line 1, Line 8), evaluated at execution time. This enables collecting atomic events per body, i.e. to distinguish between atomic events based on their source.

Referring to IQPL patterns is a special feature of our language aiming to seamlessly integrate a language for graph patterns with a language for event patterns in VEPL. This reference to the IQPL pattern is supported by the `iqPatternRef` attribute (Line 2-3, Line 9-10). The parameter list after the IQPL pattern reuses the input parameter (`B: Body`). The other parameters are not specified, as designated by their names augmented with an underscore character. (A notation similar to Prolog's anonymous predicates.) Two similar atomic events describe the cases in which the *Forward end* posture is found and lost.

Defining complex events

In the next step, atomic events are combined into a complex event. In Listing 4, the `definition` part contains the constraints for the complex event, consisting of atomic events in this specific case. The atomic events connected with the *ordered* operator (denoted with an arrow). Therefore, this pattern defines a complex event, in which the referred atomic events are observed in the specific order. Since atomic events carry information about the appropriate structural changes, this complex event will occur exactly on the series of postures depicted in Figure 2. The input parameter of the complex event (`B: Body`) and its usage in the `definition` part ensures that only atomic events originating from the same body are combined in a single complex event instance.

```
ComplexEvent ForwardGesture(B: Body){
  definition : ForwardStartFound(B) -> ForwardStartLost(B)
          -> ForwardEndFound(B) -> ForwardEndLost(B)
}
```

Listing 4: A complex event pattern reusing atomic events from Listing 3.

Complex events are built up from sequences of atomic events and other complex events, using operators of an event algebra. The event algebra of the VEPL language offers three *operators* to formalize complex event patterns: the *ordered*, the *unordered* and the *timewindow* operator. The **ordered** operator ($\mathfrak{o}$) prescribes strict ordering between the events the complex event pattern consists of. The **unordered** operator ($\mathfrak{u}$) allows the corresponding atomic events to occur in arbitrary order. The **timewindow** operator defines an upper limit for the complex event to be detected, starting from the first atomic event observed in the particular complex event pattern.

Formally, a complex event pattern $\mathcal{C}$ is built inductively from a set $\mathcal{A}$ of atomic events using three operators $\{\mathfrak{o}, \mathfrak{u}, \mathfrak{w}\}$ as follows:

- **Atomic events:** Every atomic event a is a complex event $e \in \mathcal{C}$.
- **Ordered operator:** If c_1 and c_2 are complex events then $\mathfrak{o}(c_1, c_2)$ is a complex event
- **Unordered operator:** If c_1 and c_2 are complex events then $\mathfrak{u}(c_1, c_2)$ is a complex event
- **Timewindow operator:** If c is a complex event and d is a timestamp then $\mathfrak{w}(c, d)$ is a complex event

A complex event pattern $\mathcal{C}$ is evaluated against a timestamp ordered stream of observed events denoted as $\vec{E}_0^n : e_0 \dots e_n$ with $e_i = (t_i, P_i, d_i)$ and $\forall j > i : d_j > d_i$. Initially, all e_i are atomic event instances. However, during evaluation, when a complex event instance c_j is detected after processing event e_i, then c_j is inserted into the stream (with d_i as the timestamp of the detection) to allow the detection of depending complex events later. The semantics of the operators in the event algebra is defined as follows:

- **Ordered operator:** $\vec{E}_0^n \models \mathfrak{o}(c_1, c_2)$ iff two events with types corresponding to c_1 and c_2 are present in the stream in the given order with the same parameter binding, i.e. $\exists i, j : c_1.t = e_i.t \wedge c_2.t = e_j.t \wedge e_j.d > e_i.d \wedge e_i.\sigma(P_i) = e_j.\sigma(P_j)$. The timestamp of $\mathfrak{o}(c_1, c_2)$ becomes $e_j.d$.
- **Unordered operator:** $\vec{E}_0^n \models \mathfrak{u}(c_1, c_2)$ iff both c_1 and c_2 are present in stream in an arbitrary order $\vec{E}_0^n \models \mathfrak{o}(c_1, c_2)$ or $\vec{E}_0^n \models \mathfrak{o}(c_2, c_1)$; The timestamp of $\mathfrak{o}(c_1, c_2)$ is $max(e_i.d, e_j.d)$.
- **Timewindow operator:** $\vec{E}_0^n \models \mathfrak{w}(c_1, d_1)$ iff exists an event e_i in the stream with timestamp value less then d_1, i.e. $\exists i : c_1.t = e_i.t \wedge e_j.d < d_1$.

Defining transformation rules

As the final step to our approach, the actual streaming transformations are defined. Vepl enables defining model transformations and organizing them into rules guarded by the previously defined complex event patterns. In principle, an arbitrary transformation language can be used as an action language (e.g. Xtend as in our example). All variables are bound when the trigger event is instantiated are accessible in the action part. Listing 5 shows a rule containing a model transformation which executes the action defined within the `action` block on the appearance of the `ForwardGesture` pattern, referenced in the `event` block.

```
Rule transactionRule {
  event : ForwardGesture(B: Body)
  action {
    //acquiring the complex event
    val observedComplexEvent = activation.observableEventPattern
    //extracting the parameter
    val body = observedComplexEvent.B
    // additional operation to be executed
  }
}
```

Listing 5: A streaming transformation rule

3.3 Detecting complex events

The event processing algebra, its operators and logical structures are mapped to a *deterministic finite automaton* (DFA) based representation, to keep track of partially and fully matched complex event patterns. As highlighted in Figure 4, exactly one automaton is generated for every complex event pattern at compile time. *States* in the automaton represent the relevant phases of detecting the complex event pattern, i.e. the different states of the pattern matching process.

Transitions of the automaton identify how the matching process can evolve from one state to another in accordance with the operators used in the complex event pattern and the triggering event.

During execution time, *tokens* represent the (partial or complete) complex event pattern *instances* which are stored in the states of the automaton. If there is a token at a state of the DFA, and the next event in the event stream corresponds to the trigger event of an outgoing transition, then the token is passed along the transition to the next state, thus the detection of the complex event enters a new phase. There may be multiple tokens flowing in the same automaton at a time since the next event in the stream may contribute to different parts of the same complex event pattern according to its context. When a complex event is detected, a new complex event instance is placed to the event stream with corresponding type and timestamp.

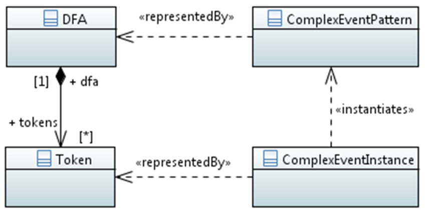


Fig. 4: Mapping between complex event patterns and the semantic model.

Event processing contexts specify constraints on how occurrences may be selected when looking for occurrence patterns that match the operator semantics [13]. Due to space restrictions, the reader is referred to [14] for the details of complex event pattern detection in VEPL. There we also prove that the automaton representing the detection cycle of complex events is always finite and deterministic.

4 Architecture and use of the prototype tooling

In this section, we give an overview of the technological aspects and the tooling of our approach. First, in Section 4.1 we present an architecture and a prototype tool VIATRA-CEP[4] for processing complex events and supporting streaming transformations. We also present the tool in action along a sample execution scenario of our case study in Section 4.2.

4.1 Architectural overview

Figure 5 presents the architecture of our streaming transformation framework. The *Model* is continuously queried by an *Incremental query engine* with queries that are defined using the *Query language*. This enables not only to efficiently obtain the match sets of a query, but it also continuously tracks changes of the model.

Changes in the model are continuously propagated to the query engine through a notification API, where callback functions can be registered to instance model

[4] `https://incquery.net/publications/viatra-cep`

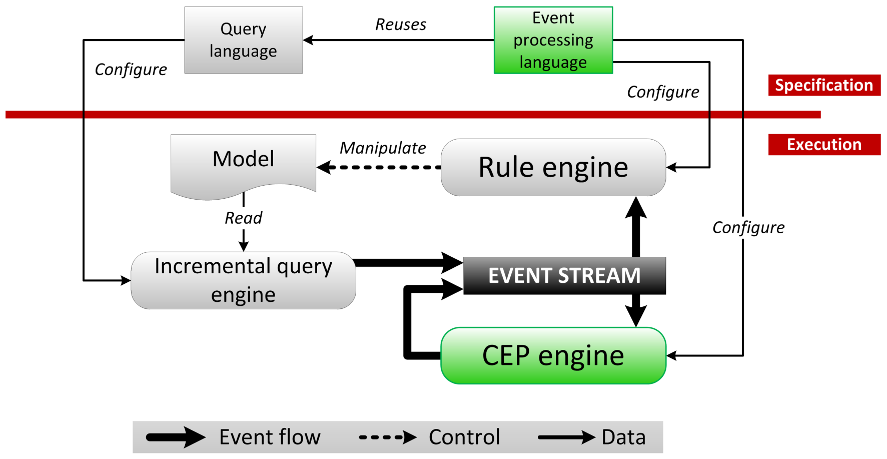


Fig. 5: Conceptual overview of the approach with our key contributions highlighted.

elements that receive notification objects (e.g. ADD, REMOVE, SET etc.) when an elementary model manipulation is carried out. The framework internally stores and maintains the partial pattern matches as notifications arrive.

As a query evaluates successfully, it produces a tuple of elements as the match set. This data is wrapped into atomic *change events* and published on the *Event stream*. The *Event stream* is continuously processed by a reactive *Rule engine*, which handles the triggering of the predefined model transformations.

In order to activate streaming transformation rules guarded by complex event patterns, the *Event stream* is also processed by a *CEP engine*. The engine continuously evaluates the complex event patterns based on the processed atomic events. If a complex event pattern is matched, a complex event instance is generated, published on the event stream and eventually processed by the Rule engine, which would trigger the appropriate model transformation.

In our prototype tool, a dedicated general purpose CEP engine (called VIATRA-CEP) was developed to support the VEPL language. However, the architecture can also incorporate the integration of an external CEP engine (such as ESPER [15]) as demonstrated in our preliminary work [4]. The case studies in [4] highlighted that significant programming overhead is required to translate structural changes to appropriate events and define complex event patterns accordingly, which requires further investigations. Our VIATRA-CEP prototype seamlessly integrates with advanced EMF-related technologies such as EMF models, the EMF-INCQUERY framework [11] for incremental queries and existing transformation languages and tools.

4.2 Sample execution of the case study

Table 1 summarizes the execution steps triggered by four consecutive snapshots of the forward gesture.

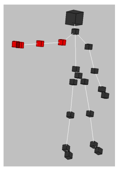
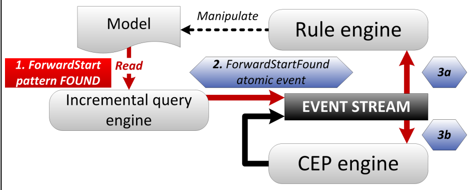
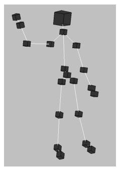
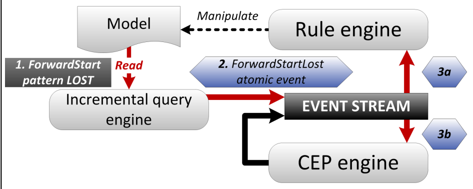
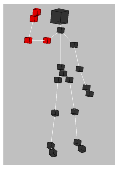
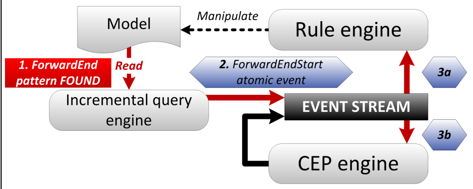
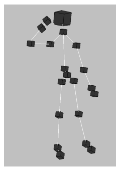
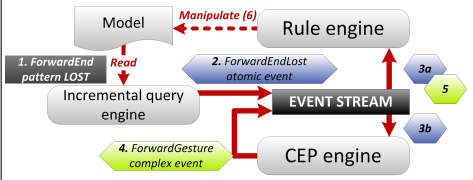
	Posture	Triggered execution
Phase #1	 FS found	 Model Manipulate Rule engine 1. ForwardStart pattern FOUND Read 2. ForwardStartFound atomic event 3a Incremental query engine EVENT STREAM 3b CEP engine
Phase #2	 FS lost	 Model Manipulate Rule engine 1. ForwardStart pattern LOST Read 2. ForwardStartLost atomic event 3a Incremental query engine EVENT STREAM 3b CEP engine
Phase #3	 FE found	 Model Manipulate Rule engine 1. ForwardEnd pattern FOUND Read 2. ForwardEndStart atomic event 3a Incremental query engine EVENT STREAM 3b CEP engine
Phase #4	 FE lost	 Model Manipulate (6) Rule engine 1. ForwardEnd pattern LOST Read 2. ForwardEndLost atomic event 3a 5 Incremental query engine EVENT STREAM 3b 4. ForwardGesture complex event CEP engine
Legend	Match FOUND Match LOST	Atomic event Complex event

Table 1: Gesture phases and the execution steps triggered.

– **Phase #1.** The *ForwardStart* pattern (Listing 1) is found *(1)* in the model by the query engine. This results in a new tuple of model elements as a match set, which data is wrapped into an atomic event by the query engine and passed to the event stream *(2)*. In *Step (3a)* the Rule engine processes the atomic event and if a transformation rule is activated, the appropriate transformation gets executed. However, since no transformation rules are associated with event *ForwardStart*, no transformation rules are activated at this point. In *Step (3b)* the CEP engine processes the atomic event as well and updates the complex event candidates, i.e. the partially matched complex events.

- **Phase #2 and #3.** In the next phase, we detect that a match of the *ForwardStart* pattern is lost. The same steps are executed as above, only this time an atomic event of type *ForwardStartLost* is published on the event stream and processed by the Rule engine and the CEP engine. In Phase #3, a *ForwardEndFound* atomic event is identified and placed on the stream.
- **Phase #4.** The *ForwardEnd* pattern is lost and a *ForwardEndLost* atomic event is published on the event stream consequently. Now there will be additional steps triggered after *Step (3b)*. After having processed the *ForwardEndLost* atomic event, the CEP engine detects the *ForwardGesture* complex event, instantiates the appropriate complex event instance consequently and publishes it on the event stream *Step (4)*. In *Step (5)* the Rule engine processes the complex event and checks for activated transformation rules. The rule defined in Listing 5 will be activated and the appropriate action will be executed in *Step (6)*.

5 Evaluation

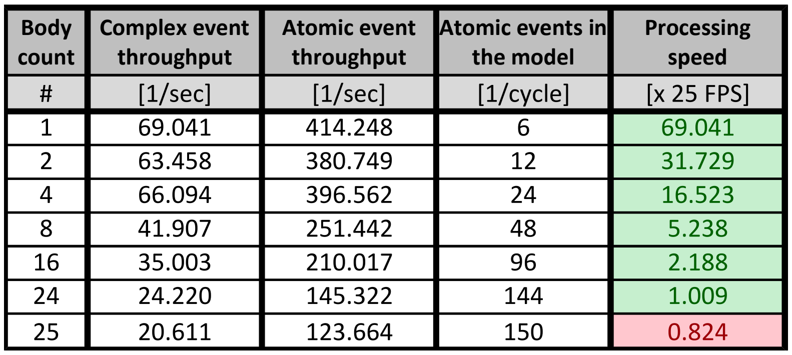
To estimate the performance and scalability of our tool, we had to design a semi-synthetic benchmark based on the use case of Section 2. The reason for this is that Microsoft KINECT can only detect at most two bodies, and the refresh rate is a fixed 25 frames per second (FPS), which is easily processed by our CEP engine.

Evaluation setup. The core of the simulation is a previously recorded real execution sequence in which the right arm is rotated. A full arm cycle consists of 12 positions, i.e. 12 frames. Every cycle yields exactly one *Forward gesture* (Figure 2) composed of the sequence of 4 atomic events; and every cycle also yields two atomic events considered as noise. This makes 6 atomic events generated for each cycle.

Our simulations aim at stress testing our CEP prototype, which is carried out by multiplying this sequence along a different number of bodies in the model. This part of the benchmark scenario is artificial in the sense that KINECT can handle at most two bodies, but the actual positions of the bodies remain realistic.

After starting the simulations, we primarily measure the *number of detected complex events per second*. From this rate, we calculate the effective processing rate (i.e. the theoretical upper limit) of the CEP engine measured in *frames per second* (FPS). This value is compared to the original FPS rate of the KINECT sensor. We continue increasing the number of bodies up to the point when the processing rate is greater than the recording rate.

Summary of results. Table 6 summarizes our results. Rows represent the individual measurements with respect to the increasing number of bodies *Body count*. The next two columns present the throughput of *complex events* (1/s) and *atomic events* (1/s), respectively. The latter is calculated from the former, since for every complex event to be detected, 6 atomic events are observed (as discussed above). The number of *atomic events in the model* denotes how many atomic events are triggered by elementary or compound model changes *per cycle*, i.e. while the right arm makes a circle. This is the number of atomic events *required* to be processed in order to achieve the frames-per-second (FPS) ratio the Kinect sensors work with. Finally, *processing speed* summarizes the FPS of our prototype compared to the basic FPS value of Kinect (25). This value is calculated as the ratio of the *Atomic event throughput* and the *Atomic events in the model*. This ratio is acceptable if it is above 1, otherwise the processing rate of complex events falls short to the data production rate of the Kinect sensor.



Body count	Complex event throughput	Atomic event throughput	Atomic events in the model	Processing speed
#	[1/sec]	[1/sec]	[1/cycle]	[x 25 FPS]
1	69.041	414.248	6	69.041
2	63.458	380.749	12	31.729
4	66.094	396.562	24	16.523
8	41.907	251.442	48	5.238
16	35.003	210.017	96	2.188
24	24.220	145.322	144	1.009
25	20.611	123.664	150	0.824

Fig. 6: Throughput and the highest processing speed.

As a summary, our measurements show that our approach scales up to 24 bodies in the model (the lowest processing speed above 1) at 25×1.009 FPS. In order to interpret this value, we need to recall that one body consists of 20 control points each of them containing 6 attributes (see *PositionedElements* in Figure 1), from which 2 are actually modified in the simulations. Therefore, for each body, 40 elementary model changes are triggered in every frame (assuming that the limbs are not reattached to different bodies).

Handling 24 bodies at a rate of 25×1.009 FPS yields approximately 24000 complex events per second. Based on our measurements (which were carried out using a 2.9GHz CPU), we conclude that our proof-of-concept implementation offers promising performance and scalability while it integrates smoothly with Eclipse based tooling. It should be noted, however, that because of the rather simple movement profile (only a few coordinates are manipulated), the results cannot be trivially extrapolated for data streams of real Kinect devices.

6 Related work

We give an overview of various approaches related to our work.

Streaming model transformations. In [1] the authors present streaming transformations working on a stream of model fragments and elements. In contrast to this technique, our approach leverages derived information regarding the model in the form of change events, which decouples the execution from the actual model. Consequently, the issues discussed in [1] (e.g. dealing with references among model elements and transformation scheduling) are not present in our case.

The concept of change-driven transformations is proposed in [2] for executing transformations on change models as input or output. Our approach extends this approach since identifying complex model changes enables CDTs of higher granularity and also enables the integration of complex event processing. Live models used in the current paper are different from living models [9], while the change pattern formalism is reused from [2], a similar formalism was proposed in [10]. A formal foundation of infinite models is presented in [16] by redefining OCL operators over infinite collections. This is complementary problem as the models themselves are finite in our case, but their lifeline is infinite due to high frequency model changes.

Complex event processing. Esper [15] is an open source event processing engine. It has been employed in our preliminary work [4], presented at the EclipseCon Europe 2012. Despite being a high-end CEP engine concerning its performance and the descriptive power of its language, supporting the scenarios like those presented in [1] is cumbersome and infeasible.

Other open CEP engines (e.g. StreamBase, Drools Fusion) can also be considered but integration into an existing MDE tooling remains a significant technical challenge since defining change patterns and feeding model (change) information into the engine requires significant programming effort. The integrated approach presented in this paper (classified as a detection-oriented CEP) overcomes this issue by providing a language supporting directly referencing graph patterns and organizing them into complex event patterns.

Processing runtime models. Processing of runtime models may introduce somewhat related challenges. Incremental model transformations are used in [17] for efficient runtime monitoring. Song et al. introduced incremental QVT transformations [18] for runtime models. However, these techniques primarily focus on obtaining a faithful model of the running system, while they do not consider event streams or complex event processing over live models.

Context-aware systems [19] introduce novel challenges for model transformations where not only business-relevant data needs to be processed, but also data from the context or environment of the system. Our approach could be a feasible solution to execute model-transformations in a context-aware fashion, e.g. in cyber-physical systems where environmental data gathered by the sensors could affect the overall transformation process.

7 Conclusions and future work

In this paper, we identified and addressed a novel class of streaming transformations [1] for live models where the models themselves are available, but they evolve at a very fast rate (resulting in thousands of changes in every second). Elementary model changes (e.g. EMF notifications) as well as derived compound changes of match sets of change patterns [2] are encapsulated into a stream of atomic events. This event stream is consumed by complex event processing techniques to identify complex series of events (appearing within a timeframe) and execute streaming transformations upon their detection.

We proposed a language built as an extension of an existing query and transformation language with execution semantics, and presented an integrated model-based complex event processing engine VIATRA-CEP to a proof-of-concept prototype. Initial experimental evaluation over a complex gesture recognition case study demonstrates the practical feasibility of our approach.

A main advantage of our framework is that models are not required to be kept in memory during transformation as only the stream of events is processed. Elementary and compound structural changes are first encapsulated into atomic changes by incremental model queries. Atomic events contain only the few relevant contextual model elements required to identify complex events and trigger related transformations for complex event processing. As a result, the time and structural dimension of changes is kept separated both from a conceptual and a tooling viewpoint.

Future work and potential applications In the future, we plan to apply the framework in various domains. *Models at runtime* (M@RT) [20] aim at representing the prevailing state of the underlying system. Processing streams of changes or change events arising from these models, instead of approaching them with batch or incremental transformations seems to be a natural fit.

Increasing the number of source models might introduce issues regarding the scalability of a transformation engine, especially when *distributed and federated data sources* are required to be handled. Dealing with change events instead of keeping model fragments from different models in memory, may significantly simplify this task.

As a primary direction for technical future work, we plan several enhancements to the change pattern modeling language, which currently lacks desirable features, such as branching patterns [21], negative patterns and temporal algebraic structures [22]. We envisage a general *canonical form* of event pattern definitions, which every event pattern could be translated into and would enable optimization steps prior to the execution.

References

1. Sánchez Cuadrado, J., Lara, J.: Streaming model transformations: Scenarios, challenges and initial solutions. In Duddy, K., Kappel, G., eds.: Theory and Practice of Model Transformations. Volume 7909 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 1–16
2. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. change (in) the rule to rule the change. Software and Systems Modeling **11** (2012 2012) 431–461
3. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
4. Dávid, I., Ráth, I.: Realtime gesture recognition with Jnect and Esper. Tech demo at EclipseCon Europe 2012, `http://incquery.net/incquery/demos/jnect` Accessed: 2014-07-01.
5. Microsoft Corp.: Microsoft Kinect official website. `http://www.microsoft.com/en-us/kinectforwindows/` Accessed: 2014-07-01.
6. Helming, J., Neufeld, E., Koegel, M.: jnect – An Eclipse Plug-In providing a Java Adapter for the Microsoft Kinect SDK. `http://code.google.com/a/eclipselabs.org/p/jnect/` Accessed: 2014-07-01.
7. Box, G., Jenkins, G., Reinsel, G.: Time Series Analysis: Forecasting and Control. Wiley Series in Probability and Statistics. Wiley (2008)
8. Eclipse Foundation: Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf/` Accessed: 2014-07-01.
9. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living Models - Ten Principles for Change-Driven Software Engineering. Int. J. Software and Informatics **5**(1-2) (2011) 267–290
10. Yskout, K., Scandariato, R., Joosen, W.: Change patterns: Co-evolving requirements and architecture. Software and Systems Modeling (2012)
11. Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Szatmári, Z., Varró, D.: An Integrated Development Environment for Live Model Queries. Science of Computer Programming (2013)
12. Eclipse Foundation: Xtext 2.6.0 Documentation. `http://www.eclipse.org/Xtext/documentation/2.6.0/Xtext%20Documentation.pdf` Accessed: 2014-07-01.
13. Carlson, J.: An Intuitive and Resource-Efficient Event Detection Algebra. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.9957` (2004) Accessed: 2014-07-01.
14. Dávid, I.: Complex Event Processing In Model Transformation Systems. Master's thesis, Department of Measurement and Information Systems, Budapest University of Technology and Economics (2013)
15. EsperTech Inc.: Esper Official Website. `http://esper.codehaus.org` Accessed: 2014-07-01.
16. Combemale, B., Thirioux, X., Baudry, B.: Formally Defining and Iterating Infinite Models. In France, R.B., Kazmeier, J., Breu, R., Atkinson, C., eds.: Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings. Volume 7590 of LNCS., Springer (2012) 119–133
17. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In Ghosh, S., ed.: Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO,

USA, October 4-9, 2009, Reports and Revised Selected Papers. Volume 6002 of LNCS. (2009) 124–139

18. Song, H., Huang, G., Chauvel, F., Zhang, W., Sun, Y., Shao, W., Mei, H.: Instant and Incremental QVT Transformation for Runtime Models. In: Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems. MODELS'11, Berlin, Heidelberg, Springer-Verlag (2011) 273–288
19. Baldauf, M., Dustdar, S., Rosenberg, F.: A Survey on Context-Aware Systems. Int. J. Ad Hoc Ubiquitous Comput. **2**(4) (June 2007) 263–277
20. Blair, G., Bencomo, N., France, R.: Models@ run.time. Computer **42**(10) (2009) 22–27
21. Ben-Ari, M., Manna, Z., Pnueli, A.: The Temporal Logic of Branching Time. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '81, New York, NY, USA, ACM (1981) 164–176
22. Gabbay, D.M.: Temporal Logic: Mathematical Foundations and Computational Aspects. Clarendon Press, Oxford (1994)