Implementing an Animated Visual λ-Calculus

Torsten Strobl[*] Mark Minas[†]

Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany

Abstract

λ-calculus is a well-known formal system for investigating computability, recursion, functional programming, etc. Reduction rules define its semantics. Several visual representations have been proposed and used for making λ-calculus more comprehensible, easier to teach, or simply more fun to use. *Alligator Eggs* [14] is an example of such a playful representation where abstraction is represented by alligators and variables by alligator eggs. *Alligator Eggs* is also an animated visual language where eating alligators correspond to function application and hatching eggs to variable substitution. This paper shows how *Alligator Eggs* can be implemented as an animated system using the diagram editor generator DiaMeta. Boolean logic is used as a running example where alligator families model boolean terms. The generated animated system allows for an animated illustration of boolean evaluation.

1 Introduction

λ-calculus by A. Church [2] is a well-known formal system for investigating computability, recursion, functional programming, etc. It has originally been invented as an abstract computing approach, and was also the origin of functional programming by inspiring Lisp. But one can also use it for representing and evaluating boolean logic.

λ-calculus is based on expressions, which can be transformed by so-called reductions to other semantically equivalent expressions. However, the textual representation of λ-calculus expressions is not very intuitive since (a) this representation of expressions is difficult to grasp, and (b) the sequence of reductions does not immediately show what is going on. This paper improves this representation with respect to these issues (a) and (b). As a solution, we propose to utilize a visual representation of λ-calculus expressions. We apply animation for making immediately clear which reductions are applied to which sub-expression with which effect.

Several visual representations have been proposed and used for making λ-calculus more comprehensible, or easier to teach. *Alligator Eggs* [14] is such a visual and

[*]Email: Torsten.Strobl@unibw.de
[†]Email: Mark.Minas@unibw.de

playful language representing λ-calculus expressions. This language has alligators and their eggs as visual components. As an example, one type of reduction process can be represented by an alligator that eats other alligators and eggs. Afterwards, the eating alligator dies, but its eggs hatch into what the alligator ate. *Alligator Eggs* thus allows for obvious and intuitive animations of the formal concept of reductions. Hence, we chose *Alligator Eggs* for visually representing and animating λ-calculus.

Meta-tools greatly simplify the process of implementing editors for a specified visual language like *Alligator Eggs*. Several meta-tools also allow for implementing transformations of diagrams which can be used for realizing the reduction process. Examples for such meta-tools are DIAGEN/DIAMETA [10], DEViL [4], GenGED [6], Tiger [1] and AToM^3 [9]. However, the support for realizing a visual editor that also allows for visualizing rather complex behavior by animations is limited. This paper describes an extension of the meta-model-based editor generator DIAMETA which allows for easy specification of visual languages with complex dynamic and animated behavior. The animated visual language *Alligator Eggs* is used as a running example although animating reductions of λ-calculus expressions is rather straight-forward.

The rest of the paper is structured as follows: The next section is a short introduction to the textual representation of λ-calculus, the visual language of *Alligator Eggs* and its dynamic behavior. Section 3 describes existing approaches in the context of specifications of animated visual languages and other visual representations of λ-calculus. Section 4 then briefly explains the existing meta-model-based editor generator and how it has been used to specify and generate the static aspects of *Alligator Eggs*. The extension of DIAMETA for animated visual languages is then described in Section 5 using the example of *Alligator Eggs*. The last section concludes the paper and reports about current work as well as plans for future work.

2 λ-Calculus

Textual λ-calculus expressions can be either a variable x, an application FG of an expression F to another expression G, or a λ-expression, i.e., an (anonymous) function $\lambda x.F$ where F is again an expression. The latter is the key concept which means a function with a (bound) variable x as parameter that may be used in F. The semantics of such expressions is defined by so-called reductions that reduce an expression to another, but semantically equivalent expression. α-conversion changes bound variables, e.g., variable x in $\lambda x.\lambda y.x$ to another, unused variable z, i.e., $\lambda x.\lambda y.x \rightarrow_\alpha \lambda z.\lambda y.z$. β-conversion applies a function to its argument by replacing each occurrence of the parameter by the argument, e.g., $(\lambda x.\lambda y.x)(\lambda z.z) \rightarrow_\beta \lambda y.\lambda z.z$. Of course, no variable in the argument may be the same as any variable in the function. Otherwise, bound variables must be changed by α-conversion first.

As mentioned previously, λ-calculus can be used in order to evaluate boolean logic. The boolean values are represented by so-called Church booleans *true* $= \lambda x.\lambda y.x$ and *false* $= \lambda x.\lambda y.y$. It is easy to verify that the representations $not = \lambda p.\lambda x.\lambda y.pyx$, $and = \lambda p.\lambda q.pqp$, and $or = \lambda p.\lambda q.ppq$ actually are suitable definitions, i.e., reducing expressions that represent boolean terms is equivalent to evaluating the represented

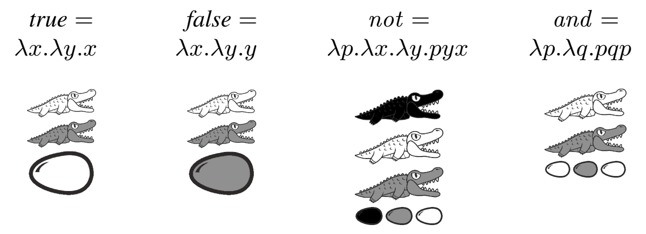

true = $\lambda x.\lambda y.x$
false = $\lambda x.\lambda y.y$
not = $\lambda p.\lambda x.\lambda y.pyx$
and = $\lambda p.\lambda q.pqp$

Figure 1: λ-expressions modeled by alligator families.

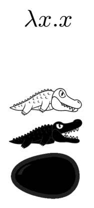


Figure 2: Old alligator

boolean terms[1]. An example evaluation is:

$$\begin{aligned}
\textit{and false true} &= (\lambda p.\lambda q.pqp)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)\\
&\to_\beta (\lambda q.(\lambda x.\lambda y.y)q(\lambda x.\lambda y.y))(\lambda x.\lambda y.x)\\
&\to_\alpha (\lambda q.((\lambda x.\lambda y.y)q(\lambda x.\lambda y.y)))(\lambda u.\lambda v.u)\\
&\to_\beta (\lambda x.\lambda y.y)(\lambda u.\lambda v.u)(\lambda x.\lambda y.y)\\
&\to_\beta (\lambda y.y)(\lambda x.\lambda y.y)\\
&\to_\beta \lambda x.\lambda y.y\\
&= \textit{false}
\end{aligned}$$

The visual λ-calculus language *Alligator Eggs* has hungry alligators, old alligators, and eggs as visual components. Let E be an arbitrary λ-calculus expression and $\langle E\rangle$ the representation of E in *Alligator Eggs*. $\langle E\rangle$ is a collection of visual components that are arranged in a tabular shape. A variable x is represented by an egg $\langle x\rangle$ where the egg color corresponds to the variable's identifier x, i.e., different variables have eggs of different color. The application FG of two expressions F and G with their visual representations $\langle F\rangle$ resp. $\langle G\rangle$ is drawn as $\langle F\rangle$ and $\langle G\rangle$ side by side with $\langle F\rangle$ left of $\langle G\rangle$, but aligned at their top. A λ-expression $\lambda x.F$ is represented by an hungry alligator (with open mouth) and the visual representation $\langle F\rangle$ of F. The color of the hungry alligator corresponds to the identifier x, i.e., every egg in $\langle F\rangle$ that represents x has the same color as the hungry alligator. The hungry alligator is drawn on top of $\langle F\rangle$; its width is the same as the width[2] of $\langle F\rangle$. Altogether, they model a so-called "family". Figure 1 shows examples of expressions for boolean logic and how they are represented in *Alligator Eggs*. Colors have been replaced by shades of gray and hatching. Expressions in parentheses, (F), are represented by an old alligator (a white alligator with closed mouth) and $\langle F\rangle$. Again, the alligator is drawn on top of $\langle F\rangle$ with the same width as $\langle F\rangle$. It is said that the old alligator "protects" $\langle F\rangle$. An example is shown in Figure 2.

β-conversion is translated into the *eating rule* in *Alligator Eggs*: The hungry alligator on top of $\langle\lambda x.F\rangle$ in an application $\langle(\lambda x.F)G\rangle$ "eats" the family $\langle G\rangle$, i.e., $\langle G\rangle$ gets deleted from the diagram. Then the hungry alligator dies, leaving $\langle F\rangle$. However, each

[1]Please note that application in λ-calculus is left-associative, i.e., $EFG = (EF)G$.

[2]This is an extension of the original description [14] of *Alligator Eggs* in order to make the language unambiguous.

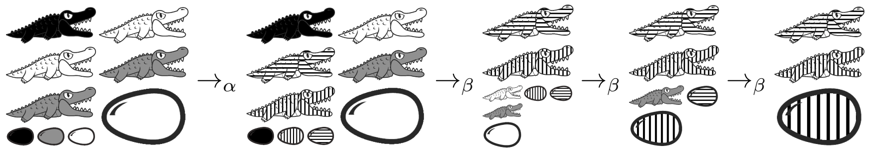


Figure 3: Evaluation of *not true* $= (\lambda p.\lambda x.\lambda y.pyx)(\lambda x.\lambda y.x)$ in *Alligator Eggs*.

egg in $\langle F \rangle$ with the same color as the died alligator gets replaced by $\langle G \rangle$. α-conversion is also necessary in *Alligator Eggs*. This is realized by the so-called *color rule* that appropriately changes the color of each hungry alligator and its eggs in $\langle \lambda x.F \rangle$ if this color also occurs in $\langle G \rangle$.[3] Finally, the *old age rule* defines the semantics of old alligators: An old alligator (representing parenthesis in *Alligator Eggs*) dies as soon as there is only a single component directly below the old alligator (e.g. like in Figure 2). A longer example, the evaluation of *not true* in *Alligator Eggs*, is shown in Figure 3.

3 Related Work

There are several tools supporting the generation of editors from visual language specifications and meta-models [6, 9, 1, 4]. However, only few of them allow animation specifications or the creation of animated editors in general. Most tools or common approaches supporting animation specifications are very limited, e.g. there is no possibility for interaction during animation, the specification of concurrent animation steps is complicated or impossible, or flexibility is missing. Also older versions of DiaMeta rudimentarily support animations [11], but practically this only means that diagram (state) changes, especially position changes, can be interpolated.

Some of the listed limitations are attributed to the utilization of transformation rules in simulation and animation, because these transformations must basically be considered as atomic operations. Therefore, a lot of efforts are put into the investigation of transformations with specified timed behavior. Transformation rules could contain a (conditional) duration and further mechanisms like interruptibility. In articles like [7] and [8] these topics are described in more detail. In [12] a graphical notion is shown, and in [13] also an event-based approach is presented.

An exemplary generator system, which - similar to DiaMeta - also applies graph models and transformations, is GenGED [6]. The system not only allows the implementation of visual language editors, but also to write (rule-based) simulation specifications. The visualization of the simulation - in this case called animation - can be specified separately, so this visualization can have a completely different layout compared to the visual language itself. In this way, the animation can be presented in another domain-specific layout, for which the term *animation view* has been introduced. However, GenGED editors cannot show such views. Instead, an automatically running "movie" has to be exported.

Next to *Alligator Eggs*, there are also other visual representations for λ-calculus. Vex uses circles for representing λ-expressions and variables [3]. Parameters are rep-

[3]This *color rule* is a bit more specific that in [14].

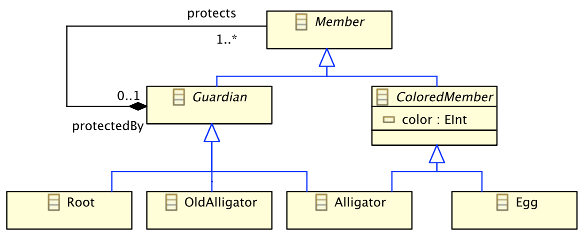


Figure 4: Architecture of a diagram editor based on DiaMeta.

resented by internally tangential circles, application by externally tangential circles. The binding of variables is explicitly represented by connecting lines.

4 DiaMeta

DiaMeta is a framework together with a specification tool for generating diagram editors from a specification [10]. The abstract syntax of a diagram language has to be specified as a meta model based on EMF [5]. Figure 4 shows the meta model for *Alligator Eggs* which comprises the composite pattern: each expression as a diagram represents an (expression) tree. *Guardian* comprises a composite node and represents an object with a (horizontal) sequence of sub-diagrams below. Concrete sub-classes are *Root* that represents a complete diagram, *Alligator* for hungry alligators, and *OldAlligator* for old alligators. Each instance of these classes contains a sequence of protected objects. Instances of *Egg*, which represents alligator eggs, are the leaves of this composite pattern. The *protects* association is ordered from left to right (not shown in Figure 4). Instances of this meta model, hence, uniquely represent *Alligator Eggs* diagrams.

Each editor generated by DiaMeta is a free-hand editor, i.e., the user may arrange visual components freely on the screen. The specification must contain descriptions of all required visual components. For *Alligator Eggs*, these are hungry and old alligators as well as eggs. When the editor user arranges such diagram components on the screen, the editor has to check whether the arrangement is a correct diagram, and, if so, what its syntactic structure is. Each editor generated by DiaMeta uses a generic architecture for solving this problem, see Figure 5: The editor consists of a drawing tool which is used by the editor user for arranging the diagram components on the screen. The arrangement is then internally represented by a so-called *graph model* as a homogeneous representation which can be used for all diagram languages. Figure 6(b) shows the *graph model* for the simple *Alligator Eggs* diagram in Figure 6(a). Each diagram component is represented by a *component node*, here *alligator*, *egg*, and *root*. The latter represents an invisible component representing the whole canvas. The specification describes how each diagram component is represented: Each component has a certain number of attachment areas. The canvas and eggs have a single attachment area, alligators have two, the alligator shape itself, and the area from the alligator to the

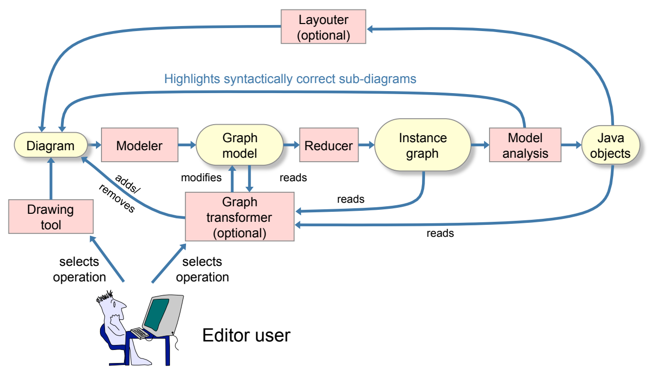


Figure 5: Architecture of a diagram editor based on DiaMeta.

bottom of the canvas. Each of these areas is represented by an attachment node that is connected with its component node by an edge with labels *canvas*, *shape*, or *below*. A *relation edge* connects two attachment areas that are related in a specific way. E.g., a *protects* edge connects the corresponding nodes if an alligator or egg lies underneath another alligator.

The graph model may grow quite large. E.g., a stack of n alligators requires $O(n^2)$ *protects* edges. The reducer (see Figure 5) transforms this graph into the *instance graph* by applying reducer rules in the specification. They are omitted here since they are not crucial for the setting of this paper. The obtained instance graph represents an instance of the specified meta model if the diagram is syntactically correct. This is checked by the model analysis.[4] Model analysis provides feedback to the user about diagram parts that are not syntactically correct by highlighting those diagram components. Model analysis also instantiates the EMF model that implements the meta model. This data structure can then be used by an automatic layout facility for beautifying or layouting the diagram.

Free-hand editing is complemented by (optional) graph transformation rules that utilize a graph transformation facility. Graph transformations may be specified for "implementing" complex diagram modifications that are triggered by the editor user. However, such transformations that operate on the graph model, but that may use information from the instance graph, too, are also a helpful mechanism for animation support.

5 Animated Alligator Eggs

This section describes how the animated aspects of *Alligator Eggs* can be specified. First the desired animations are described, and subsequently the different animation states are identified. Based on such states the used animation approach is explained,

[4]Model analysis is actually more sophisticated. Not all classes must have been determined by the reducer rules. Model analysis uses constraint solving techniques for identifying undetermined classes [10].

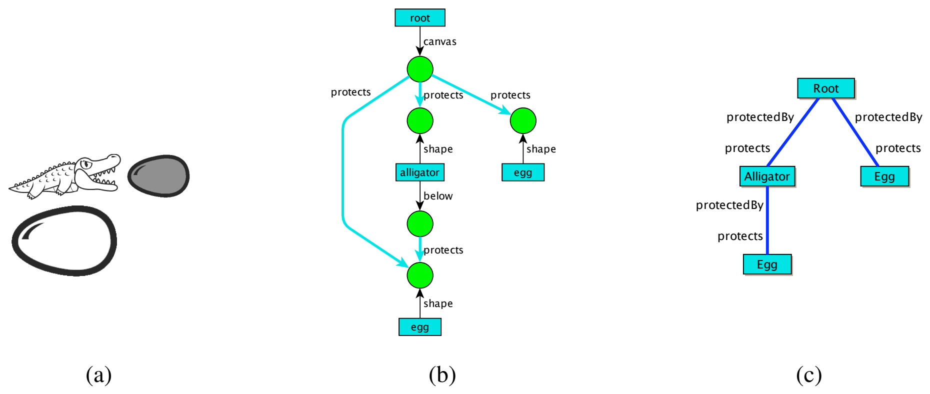


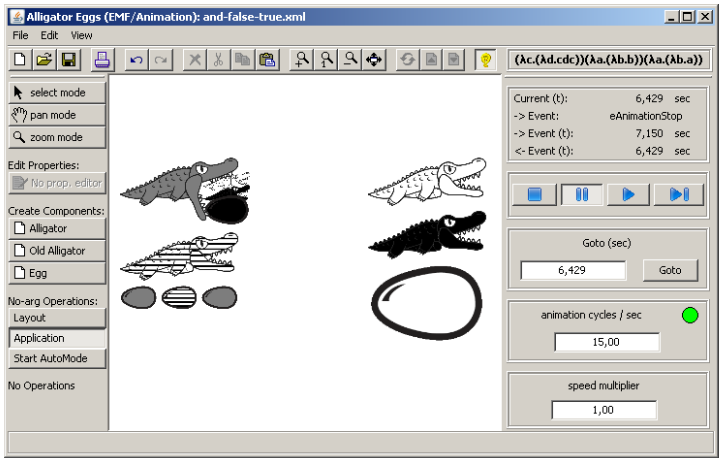
Figure 6: (a) Visual representation of $(\lambda x.x)y$, (b) graph model, and (c) instance graph.

including an event-based strategy. Finally, the concept is compared with common approaches of other frameworks, which are similar to DIAMETA.

5.1 Animation Description

Section 4 has described in short how the static part of *Alligator Eggs* can be specified. As a result, a fully functional editor for static *Alligator Eggs* diagrams can already be generated. It has also been mentioned that DIAMETA supports the specification of graph transformations. The step semantics of *Alligator Eggs* (see Section 2), can be implemented by utilizing these transformations. Indeed, only one, but rather complex graph transformation is required for this. The operation has to analyze the graph, decide on the next applicable rule and finally arrange modifications in order to get the results after the step. By triggering such a transformation the editor's user can watch the conversions step-by-step. However, users cannot follow the reduction process in more detail, which could be crucial for a better understanding of *Alligator Eggs* and the λ-calculus. Therefore, it is desirable to generate an editor which also shows internal processes by animating them. Each individual part of a rule application shall be visualized in a movie-like fashion. Again, the description in [14] was used as orientation.

The *eating rule* is split into subparts. First, the alligator eats the family in front of him. Therefore, the family moves towards the alligator's mouth. Meanwhile, the alligator is snapping, and it is also meaningful to decrease the victim's overall size. Afterwards, the alligator dies, so the shape actually rotates until the alligator is lying on its back, and it disappears by shrinking. Intermediately, the "reborn family" will hatch out of the according eggs. This means, that the egg cracks and the families appear. This way, the user can track the way of the eaten family, and also the alligator, which causes the action, can be identified. In order to have enough space for the new structure the whole diagram is also re-layouted. The *old age rule* is applied similarly. The old alligator dies and the diagram is layouted (both overlapping in time). Finally, during the *color rule* affected components are recolored, which is transacted by cross-



(λc.(λd.cdc))(λa.(λb.b))(λa.(λb.a))

```
Current (t):        6,429    sec
-> Event:           eAnimationStop
-> Event (t):       7,150    sec
<- Event (t):       6,429    sec
```

Goto (sec): 6,429

animation cycles / sec: 15,00

speed multiplier: 1,00

Figure 7: Screenshot of the *Alligator Eggs* editor while alligator is eating

fading their colors. In Figure 7 an exemplary scene while processing the eating rule is illustrated.[5]

5.2 Animation States

The previous description already indicates that required animations can be separated into multiple, also concurrent, phases. The idea now is to encode informations about currently running phases (along with possible parameters) within the diagram's state or even the state of individual components or component groups. Specific state transitions then imply the transition from one phase to the next. Figure 8 shows a possible translation of the textual animation description (see Section 5.1) into a state machine diagram, even with more details like the duration of the individual phases. In particular, it depicts the states of one resp. two families during the execution of a rule, and how individual members are animated in the meantime. In the lower left corner the diagram's start state *Static* can be found, the editor must not show animation for involved diagram components here. If a rule shall be applied, the state switches to state *Animated*, if possible. Before this transition, the system actually has to determine the applicable rule as shown (also with according priorities).

In the following, the most complex substate of *Animated* will be exemplified: *Eating Rule*. In this state the associated rule with the same name is processed. Directly after the rule is applied, involved components pass through the *Eating Phase*, also a substate. 3 sec. afterwards, the state automatically passes over into the *Rebirth Phase*. Concurrently, the dying alligator and the animated re-layout process are shown during this phase. All substates again can pass over after 3 sec., however, the re-layout process is delayed by 1.5 sec., so actually 4.5 sec. are required.

[5] An animated example can be found at: http://www.youtube.com/user/diametaanimated

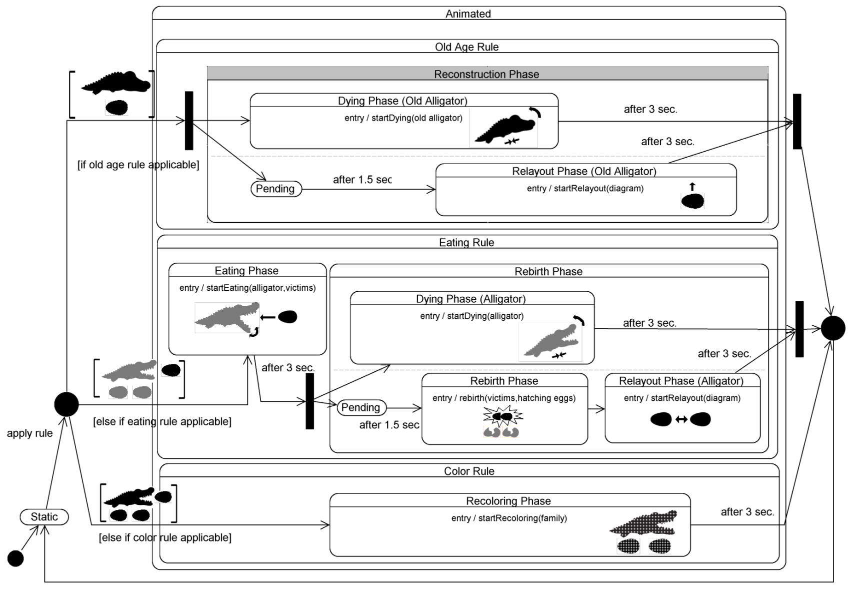

Animated
Old Age Rule
Reconstruction Phase
Dying Phase (Old Alligator)
entry / startDying(old alligator)
after 3 sec.
after 3 sec.
[if old age rule applicable]
Pending
after 1.5 sec
Relayout Phase (Old Alligator)
entry / startRelayout(diagram)
Eating Rule
Eating Phase
entry / startEating(alligator,victims)
Rebirth Phase
Dying Phase (Alligator)
entry / startDying(alligator)
after 3 sec.
after 3 sec.
after 3 sec.
Rebirth Phase
entry / rebirth(victims,hatching eggs)
Relayout Phase (Alligator)
entry / startRelayout(diagram)
Pending
after 1.5 sec
[else if eating rule applicable]
apply rule
Color Rule
Recoloring Phase
entry / startRecoloring(family)
after 3 sec.
Static
[else if color rule applicable]

Figure 8: State diagram - rule application

The example has shown the definition of *animation states* for a group of components, but such states can also be defined for individual components. For instance, each component can be in state *Static* or *Animated*. The *Animated* state itself can be separated into multiple substates: *Rotating*, *Shrinking*, *Moving*, *Snapping* (only alligators), *Hatching* (only eggs), etc. Again, concurrent states are possible, e.g., if an alligator is rotating and shrinking. Figure 9 shows a timing diagram, which outlines the states of individual diagram components while processing the *eating rule*.

5.3 Animation Concept

The way a visual language like *Alligator Eggs* is presented is specified by its *concrete syntax* (in contrast to the abstract syntax, e.g., given by the meta-model). In DIAMETA this syntax can be specified for each visual component. Until now, the possibilities have been designed for specifying non-animated (static) components. Considering this specification and the internal *graph model* together with attributes of component nodes, e.g., the x and y-position of the component, the editor is able to draw the static visual representation. However, this basic approach is not necessarily limited to draw static diagrams, if time is considered for the mapping of the graph to its representation. Hereby, the time t could be a given parameter for the whole drawing process. Other *animation parameters* can be stored within component nodes in exactly the same way as parameters for the static representation. Possible animation parameters would be a given start time, a start position or a constant velocity. With these informations a

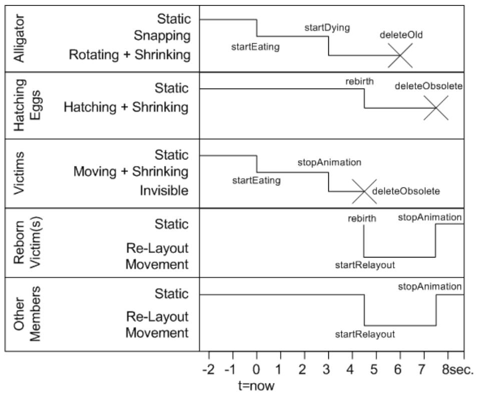


Figure 9: Timing diagrams - *eating rule*

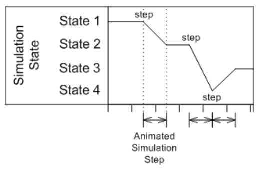


Figure 10: State transitions consuming time

component can be visualized at different x-positions depending on the drawing time t. In an animation sequence with continuously increasing t and a periodically redrawn diagram, this results in a smoothly animated motion. However, animation parameters are not limited to simple values in the affected component node. Also the relations to other components and their attributes can be taken into account, e.g. an attached arrow component which specifies the movement direction.

In order to implement *Alligator Eggs*, we chose this approach and extend the DiaMeta framework by adding an animation package. This package provides the required functionality, e.g. the global time parameter. The specification for visualizing the animated component itself is currently a programmed block of code in order to allow flexible mathematical calculations depending on arbitrary information contained in the graph model.

Now, also the animation states described in Section 5.2 can be reconsidered. The state of an individual component can be stored within the component's attributes, and this state is the basis for deciding how the component is animated. In addition, animation parameters (also attributes) can be required for the animated visualization. For example, if an alligator is in state *Rotating* (see Figure 9), the following attributes are used: $animationStart$, $animationStop$, $angleStart$, $angleStop$. Together with the global time and a specified algorithm, e.g. linear interpolation between start/stop time and start/stop angle, the animation can be calculated. Another example is an alligator's *Eating* state in which the alligator's mouth is snapping. A simple flag $snapping$, along with $animationStart$ and $animationStop$, is sufficient in this case. The alligator's component is drawn via two images: the main body and the lower jaw. If the flag is not set, the lower jaw is drawn statically onto the rest of the body. Otherwise, it is drawn slightly rotated around the jaw's joint. Thereby, the angle is interpolated (cosine-based) between fixed values with periodic repetitions.

5.4 Events

Of course, animation parameters can be changed by the user like other attributes. Components can be dragged via mouse, or property editors can be used to change internal values. However, changing animation parameters this way does not make much sense for many use cases. A useful mechanism in order to set animation parameters, or animation states to be more general, are graph transformations, which are already available in DIAMETA. With them important values like start/stop times can be calculated automatically, and complex diagrams can be analyzed and modified in an established way. As a consequence, they can be used to change animation states. Some of these graph transformations required in *Alligator Eggs* are already shown in Figure 8. For example, if the *eating rule* shall be applied and the *Eating Phase* is entered, the *startEating* graph transformation is applied on entry, and this transformation actually sets the new animation state. Also arguments can be passed to the transformation, in this case that is the alligator and its victims. A more complex transformation is *startRelayout*. It is able to layout the whole diagram. Indeed, the functionality can be shared with the static layout specification, which is already specified (see Section 2). The difference is that the results of the algorithm are either used for the definite positions or the arrival points of linear movements.

The remaining question concerns the point in time these graph transformations, which actually control the animation flow, are executed. Therefore, it must be possible to specify *events*. Whenever an event is triggered, the according transformation is performed. DIAMETA already provides the specification of an obvious event type: *user events*. It is possible to add editor buttons, which initiate graph transformation, if clicked. However, also other unanticipated events can be considered as user events: moving components via mouse, key strokes, adding or removing components. The state transition from *Static* to *Animated* is an example of user events, because it is actually triggered, if the user clicks on the button "Application" (see Figure 7).

The second type of events are *timed events*. In Figure 8 several transitions are fired after specific time events, e.g. "after 3 sec". Therefore, the DIAMETA animation package also includes an event queue, wherein timed events must be registered. Thereby, the event times are calculated based on the graph model. For example, three seconds must be added to the point in time when the *Eating Phase* was entered in order to obtain the time of the next transition. Also more complex calculations are possible, e.g. computing the time when two components collide on their trajectory, etc. Individual component attributes as well as the whole graph structure can be taken as basis of the calculation. Again, graph matching mechanisms can help finding and calculating the times of these events. The time along with affected components, the event context, is then stored in the event queue. The queue, hence, must be revised after each change of the graph model. If components are added, removed or changed, it is also possible that new events appear and already registered events must be modified or updated. As soon as the global time t reaches the time of an enqueued event, the graph transformation that has been specified for this event type gets triggered along with the actual event context (affected components). As described, this can lead to a change of the animation state. If more than one event share the same time, the DIAMETA approach is to priorize some events by specification.

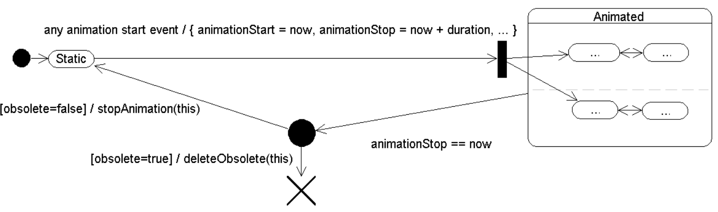

Animated
any animation start event / { animationStart = now, animationStop = now + duration, ... }
Static
...
[obsolete=false] / stopAnimation(this)
animationStop == now
[obsolete=true] / deleteObsolete(this)

Figure 11: State diagram - animation states of individual components

There is also a third event type: *immediate events*. They occur time-independently as soon as a specific state is reached, e.g. after a graph transformations. An example is the event which fires the transition from the *Rebirth Phase* to the *Relayout Phase (Alligator)*. Assuming the former state, the connected graph transformation (*rebirth*) recreates an instance of the eaten family for each hatching egg. Directly afterwards, the state changes to *Relayout (Alligator)* and the graph transformation for re-layouting, *startRelayout*, is performed.

Otherwise, the work for creating the *Alligator Eggs* event specification for starting animations based on the state machine diagram is straight-forward. However, there is still the need to stop animations after specific phases end. Of course, these animation stops could be initiated by individual events, which are triggered if a state exits. Figure 11 shows an elegant alternative, if "animation stop" always means to reset all animation parameters, no matter which component type. The component's state shall switch to *Static* automatically, if the global time t reaches a component's animation parameter $animationStop$. Regularly, this will cause the graph transformation *stopAnimation*. However, as a nice additional feature, if the flag $obsolete$ is set for the component, it is marked for deletion and can even be deleted automatically by the graph transformation *deleteObsolete* (cp. Figure 9).

5.5 Comparison With Other Approaches

As already mentioned in Section 3, animation within diagram editors, especially in the field of generator frameworks, is often used as a supplement to simulation (e.g. GenGED [6], DEViL [4]). Thereby, animation is used for user-friendly visualization of individual simulation steps. Without additional techniques each step is an atomic operation. However, its duration - which is actually 0 - must be stretched in time in order to animate changes. Figure 10 shows the chronological sequence of state transitions, which are applied this way. In the following, this animation concept is called *transition animation*. On the contrary, the *state animation* concept outlined in this paper sticks with instantaneous transitions (cp. Figure 9).

The *Alligator Eggs* editor can be generated via both approaches. Each rule application can be considered as simulation step. With transition animations a way to specify the whole animation for this single step is required. In case of the rather complex eating rule, specification possibilities must be very flexible, e.g. an arbitrary amount of reborn families must be animated. A specification following the concept of state animations applies to shorter, usually less complex animation sequences, which can be described via graph transformations and events.

Furthermore, transition animation sequences cannot overlap in time, because the underlying simulation steps themselves cannot overlap. For example, the *Alligator Eggs* rule applications - considered as simulation steps - must be executed consecutively. However, it would be possible to execute several independent rule applications simultaneously and time-shifted.[6] Components can be animated independently, and by using the event approach they could also interact with each other, e.g. collision events during a movement. Even unpredictable interactivity while running the animation is possible (*user events*), and user activities can be considered in the animated visual language's design. In case of the *Alligator Eggs* editor, it is possible to delete hatching eggs or victims during the *Eating Phase*, for example.

Finally, the state animation concept also covers the animation of diagram languages, whose underlying models cannot be simulated. Animations could always be used as eye-catcher to highlight particular diagram elements, for example.

6 Conclusions

This paper has shown that also animated editors for visual logic languages like *Alligator Eggs* can be successfully specified and generated using DiaMeta. The resulting editor can be used in order to model visual λ-expressions. Animations enable the user to track changes during the reduction of expressions.

The DiaMeta framework has been extended in order to apply the presented animation strategy. This strategy differs from other approaches widely spread in the area of meta-tools and editor generator frameworks. Even some limitations of other approaches are resolved. However, the specification complexity of animations and events must still be improved. Currently, the concrete visualization and calculations must be written by hand. Patterns for different animation types and processes including their parameters would be desirable, even if loosing flexibility. With such patterns, it is also more likely, that higher-level cartoon-like animations (e.g. by using predefined effects), which are especially suitable for an editor like *Alligator Eggs*, are used instead of simple interpolated transformations. The approach should also be investigated with regard to different types of animated visual languages. Perhaps certain specification procedures, e.g. using state machine diagrams, can be identified and described in order to simplify overly complex specifications. Finally, the applied strategy also seems to be suitable for specifying highly interactive animated languages.

References

[1] E. Biermann, C. Ermel, J. Hurrelmann, and K. Ehrig. Flexible visualization of automatic simulation based on structured graph transformation. In *VLHCC '08: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 21–28, Washington, DC, USA, September 2008. IEEE Computer Society.

[6]This feature has been realized with the concept presented in this article

[2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:354–363, 1936.

[3] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *VL '95: Proc. 1995 IEEE Symp. on Visual Languages*, pages 294–301, Darmstadt, Germany, Sep 1995. IEEE Computer Society Press.

[4] B. Cramer and U. Kastens. Animation automatically generated from simulation specifications. In *VLHCC '09: Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, Corvallis, Oregon, USA, September 2009. IEEE Computer Society.

[5] EMF – Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/, 2009.

[6] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Tech. Univ. Berlin, Fak. IV, Books on Demand, Norderstedt, 2006.

[7] S. Gyapay, R. Heckel, D. Varro, and D. Varr. Graph Transformation with Time: Causality and Logical Clocks. In *ICGT '02: Proc. 1st Int. Conf. on Graph Transformation*, pages 120–134. Springer-Verlag, 2002.

[8] H.-J. Kreowski and S. Kuske. Graph transformation units with interleaving semantics. *Formal Asp. Comput.*, 11(6):690–723, 1999.

[9] J. d. Lara and H. Vangheluwe. AToM3: A Tool for Multi-formalism and Metamodelling. In *FASE '02: Proc. 5th Int. Conf. on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[10] M. Minas. Generating meta-model-based freehand editors. In *GraBatTs '06: Proc. 3rd Int. Workshop on Graph Based Tools, Satellite event of 3rd Int. Conf. on Graph Transformation*, Natal, Brazil, September 2006.

[11] M. Minas and J. Gottschall. Specifying animated diagram languages. In *TVL '97: Proc. Workshop on Theory of Visual Languages*, Capri, Italy, 1997.

[12] J. E. Rivera, C. Vicente-Chicote, and A. Vallecillo. Extending visual modeling languages with timed behavioral specifications. In *IDEAS 2009: Proc. 12th Iberoamerican Conf. on Requirements Engineering and Software Environments*, pages 87–100, Colombia, April 2009.

[13] E. Syriani and H. Vangheluwe. Programmed Graph Rewriting with Time for Simulation-Based Design. In *ICMT '08: Proc. 1st Int. Conf. on Theory and Practice of Model Transformations*, pages 91–106, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] B. Victor. Alligator eggs! A puzzle game. http://worrydream.com/AlligatorEggs/, May 2007.