Bachelorarbeit

Visual Editor for CEP

Maija Heckel
heckelm@cip.ifi.lmu.de

Bearbeitungszeitraum: 5. 5. 2011 bis 22. 09. 2011
Betreuer: Simon Brodt Lehrstuhlmitarbeiter
Verantw. Hochschullehrer: Prof. Dr. François Bry
Zusammenfassung

Abstract
Aufgabenstellung

Kopie der Original-Aufgabenstellung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, September 14, 2011
## Contents

1 Introduction .......................................................... 1

2 Running Examples ..................................................... 2

3 Concepts of the visual language .................................... 5
   3.1 Event type ................................................................ 5
   3.2 State type ................................................................ 5
   3.3 Query box ............................................................ 6
      3.3.1 Input box .......................................................... 6
      3.3.2 Output box ....................................................... 8
      3.3.3 Body box .......................................................... 8
   3.4 Stream edges ......................................................... 9
      3.4.1 Input edges ....................................................... 11
      3.4.2 Output edges .................................................... 11

4 Visual Editor .............................................................. 14
   4.1 Zoom .................................................................... 14
   4.2 Folding ................................................................... 14
   4.3 Filter ..................................................................... 15
   4.4 Dynamic interaction ............................................... 16

5 Interaction ................................................................. 18
   5.1 Adding ................................................................... 18
   5.2 Deletion .................................................................. 22
   5.3 Editing .................................................................... 24
   5.4 Positioning and movement ....................................... 24
   5.5 Selection .................................................................. 25
   5.6 Folding .................................................................... 27
   5.7 Error indication ...................................................... 28

6 Implementation of the visual editor in Java .................... 29

7 Related Work ............................................................. 30
   7.1 Visual editors of the same research group ................. 30
      7.1.1 visXcerpt .......................................................... 30
      7.1.2 visKWQL .......................................................... 31
   7.2 Visual editors for the modelling of simple business applications ...... 31
      7.2.1 Visual Rules Modeler ......................................... 31
      7.2.2 Drools Flow ..................................................... 32
   7.3 Visual editors for Complex Event Processing ............... 32
      7.3.1 SYBASE ........................................................... 32

8 Conclusion ................................................................. 33
1 Introduction

This bachelor thesis presents the concept of a visual editor for the Complex Event Processing language DURA [8]. Complex Event Processing makes it possible to derive complex events from a large number of so-called simple events. The aim is to gain higher-level knowledge from these simple events, while they are actually appearing [3]. A sensor, which measures the temperature of a specific area in periodical time intervals, for example, can create simple temperature events from these measurements. Simple events arrive in the form of a stream, which flows through different queries. These queries define kind of patterns, matching combinations of events that fulfill certain conditions on the relations between the events. The conditions are defined with respect to the carried data of the events and particularly to timestamps, associated with these events. If a combination of required events matches the pattern defined in a query, the query derives a new complex event. Complex Events, once derived, can then be the input for other queries, which generate further complex events, and so on.

The programming language DURA requires hierarchical dependencies between the queries, which is a usual restriction in Complex Event Processing. DURA is a declarative event and action language for reactive emergency management and was designed for the EMILI (http://www.emili-project.eu/) project. The aim of DURA is to help emergency experts in writing rules, for example, the evacuation procedure in case of emergency, in an automatically or semi-automatically way. DURA is designed to be easier to learn and to handle, than other Complex Event Processing languages and can therefore be used by non computer scientists, as well. DURA offers two kinds of rules, namely deductive and reactive rules. Deductive rules derive complex events from simple ones without having any side-effects. Reactive rules are used to trigger actions based on the observed events. Reactive rules usually have side-effects.[8]

As mentioned before, this bachelor thesis presents a visual editor for DURA, called visDURA. The goal of the editor is to further reduce the complexity of usage, i.e. to make it even easier for the emergency experts to formulate Complex Event rules. With this visual editor the user does not have to write that much code, instead he deals with a graphical user interface. This interface mostly provides the structure of the queries and thereby prevents syntax errors. To achieve the structure of the queries the editor provides different types of graphical elements. While placing and editing these graphical objects on the screen, he can develop a whole DURA program more easily. Additionally to the given structure the editor provides further guidance in form of hints and warnings for the user. Thus the developer is able to learn the handling of the visual editor while using it directly. As a first step, this bachelor thesis deals with the conception and implementation of the deductive rules. Reactive ones are left for future work.

The following text of the bachelor thesis provides a deeper insight into the underlying concepts of the visual editor for DURA and the main parts of the implementation in Java. Chapter 2 deals with a short introduction to some examples of the DURA language, which are necessary to understand the concepts of the visual editor in the later text. Section 3 to 5 form the main part of the bachelor thesis and present the three working directions towards a visual editor for DURA. Section 3 describes the visual language with its particular components, Section 4 introduces the actual visual editor and its improvements compared to other existing editors for Complex Event Processing. Finally, Section 5 deals with different interaction forms for the visual editor, which will increase the ease of use for the developers. Section 6 then provides a short look at the main parts of the implementation in the programming language Java, with the used eclipse frameworks EMF (http://www.eclipse.org/emf/) and Graphiti (http://www.eclipse.org/graphiti/) and Section 7 presents existing visual editors for Complex Event Processing, as well as graphical editors.
in other areas, and compares them to the here introduced editor for DURA. Section 8 of
the bachelor thesis gives an outlook of the next steps concerning the development of the
visual editor.

2 Running Examples

In the following, some DURA rules will be explained, which serve as example for
introducing the visual representation of the DURA language (see [8]). A deductive
rule of the DURA language consists of a head and a body. The head defines
the derived complex event with its corresponding attributes and starts with the key-
word “DETECT”. The body defines the events and their attributes, which in com-
bination match the query and therefore derive the complex event. The body part
starts with the keyword “ON” and ends with “END”. Optionally conditions can be
defined at the end of the body, which are introduced with the keyword “where”.

DETECT
    high-temp{ id{ var Id }, area{ var A } }
ON
    and{
        event i: temp{
            id{ var Id },
            area{var A},
            temperature{var T1} },
        exists event j: avg-temp{
            area{ var A },
            temperature{ var T2 } }
    } where{ j during from-begin-backward(i, 30 sec),
        var T1 >= 80, var T2 >= 50 }
END

This query derives a high temperature event for a certain area, if additionally to
one measured high temperature value a high average temperature in this area has been
measured within the 30 seconds preceding the high temperature measurement. Thus, two
events temp and avg-temp build the input for this specific query and have to achieve
several conditions. In this specific example the temperature value of the temp event has
to exceed 80 degrees and the temperature value of the avg-temp event must rise about 50
degrees in the last 30 seconds in the same area. Moreover, the query is not interested in
the concrete avg-temp event, but only wants to check, if it took place (expressed with the
keyword exists).

DETECT
certain-alarm{ area{ var A } }
ON
    and{
        event i: high-temp{ area{ var A } },
        event j: smoke{ area{ var A } }
    }
END

In this DURA example the previously derived high-temp events are used as input for
another query. If both, a smoke and a high-temp event occur in the same area then the
rule derives a certain-alarm event for that area.
The last DURA example derives a sensor-broken event, if a temperature sensor (identified by the id attribute) delivers a temp event, which is not followed by another temp event within 12 seconds, i.e. a sensor-broken event is derived if the sensor pauses or stops sending events.
3 Concepts of the visual language

This section describes the individual concepts of the visual language for DURA rules. Later parts of the bachelor thesis will deal with the visual editor and different interaction features for it.

3.1 Event type

An essential concept of DURA and therefore of the visual language is the event type. An event type has a unique name and an arbitrary number of data attributes. In the visual representation, the event type has a rectangle shape and consists of two parts: the first represents the name of the type, the second the corresponding attributes. The attributes are related to a data type. The syntax for attribute specifications follows the fixed pattern ATTribUTENAME: TYPE, which is adopted from the Unified Modelling Language (UML). Figure 3.1 shows the visual representation of the event type temp, with its data attributes id, area and temperature. The representation of an event type definition occurs only once in the whole diagram. The intention is to depict the flow of events through the queries. This becomes more clear in Section 3.4.

3.2 State type

A state type, or a statefull object type, as it is called in DURA, is similar to an event type. A state type can hold data attributes representing the actual state. These are similar to the attributes of an event type. In the visual language a state type is also represented by a rectangle, but has a dotted border. Consequently it can be distinguished from the event type. The head of its graphical representation contains the name of the state type, the second part consists of the attribute specifications. In the example given in Figure 3.2 the name of the state type is operation-mode whose current mode is represented by the emergency attribute. A current mode is like an ordinary data attribute but may change over time. However, this is done by reactive rules which are not part of this bachelor thesis. Unlike the representations of event types, the representations of state types occur as often as their actual state type is used in the diagram. However, they have only to be specified once and all representations are kept synchronized with their definition. Once specified they can be reused at any time they are needed for a query. All in all, in contrast to the representation of an event type the representation of one specific state type may occur for several times in the diagram, namely one for every query using this state type. The reason for the different treatment of event type and state type definitions is that states are not part of the data flow. For this reason changes of states are caused by reactive
3.3 Query box

Event types and state types are two important concepts of DURA, but they do not describe how the data is used to derive new information. This is done by deductive rules. Basically, deductive rules consist of two parts: the body, which is a query and the head, where the derived event and the values of its data attributes are described. Thus we have to introduce another central concept of the visual language, namely the query box. The query box represents a query over appointed events and states and is therefore connected to the corresponding types (compare 3.4). A query box is visualized as a rectangle with rounded corners and contains three nested boxes. These nested boxes are the input area, where a defined number of input boxes is placed, the output box and the body box, as you can see in Figure 3.3. In the following, these three elements are described in more detail.

3.3.1 Input box

The first nested box is called input box and corresponds to a literal in the DURA rule. An input box specifies that any match of the query requires an instance of the given event or state type. The input box extracts attribute values from the events or states and binds them to variables. These variables can then be used in the rest of the query box (more information about variable binding may follow in the later text). In the input box the symbolic binding to the variables takes place, which is independent from the instance. The real binding happens at runtime. A query box may contain an arbitrary number of input boxes (at least one). An event type can be the source for more than one input box of the
3 CONCEPTS OF THE VISUAL LANGUAGE

3.3 Query box

Figure 3.4: Heads of a negated and an existential input box, which are part of the query box

Figure 3.5: Explicit binding of a variable A on the attribute area inside the input box

same query box. A representation of a state type can be used only once, consequently it corresponds to exactly one input box. All input boxes in one specific query box are associated in a conjunctive manner. Beside the event and or state type the head specifies an identifier for the input. The full specification in the head has the form IDENTIFIER: TYPE. In Figure 3.3 one identifier is i and the corresponding event type is smoke. The representation of an event type in a query and therefore the whole input box can appear in a positive, negated or existential way. The heads of the two special types, negated and existential, are shown in Figure 3.4. The semantic for those types is the same as in DURA [8]. Thus, if a query asks for the existence or non-existence of an event, it only checks, if the event occurs or not occurs. In this case no data will be extracted from the events and no variables will be bound in the query box.

There are two ways to use data attributes of events or states in the body and output box of a query. The first possibility is to bind variables to the attributes within the input box (see Figure 3.5). After that it is possible to use the variable anywhere in the query box, whenever the specific attribute of the event is needed. This method is called “explicit variable declaration” and is achieved via VARIABLE = ATTRIBUTE. If the same explicit variable is defined in different input boxes of the same query, the variable has to be bound to the same value at each time. That comes up to a test of equality.

The other possibility is the implicit variable declaration. These variables are defined by default. The basic idea is that every attribute can be unambiguously be identified by the name of the attribute and the identifier from the head of the input box. These two components, separated by a dot, define an implicit variable, which can be used in other parts of the query box. With this method the attribute also has a unique name dependant from the actual input event inside the query box. Figure 3.6 shows a pseudo definition of

Figure 3.6: Pseudo definition of an implicit binding of a variable inside the input box. The term i.area can be used to access the attribute area of the i:temp event inside the query, therefore it is similar to a variable.
an implicit variable declaration with the variable \( \text{i.area} \) and the attribute \( \text{area} \). In reality such a declaration is not necessary because it is provided by default, as mentioned before. However, with these two different ways of declaration the user has the chance to choose the method he prefers and the opportunity to mix both concepts inside a single query box.

### 3.3.2 Output box

The second type of nested boxes is the output box. Every query box has exactly one output box. This box is associated with the output event type, and therefore shows the name of the type (:OUTPUTEVENT) in the head of the box. If there is more than one output box corresponding to the same event type, then the output of the boxes is combined in a disjunctive manner. That means, every query connected to the event type is able to derive events of this type, but the queries do not have to be matched at the same time to generate the new event. Inside the output box you can assign values from the previously defined variables to the data attributes of the derived event. The binding is the other way round as inside an input box (\( \text{ATTRIBUTE} = \text{VARIABLE} \)).

### 3.3.3 Body box

The body box (compare Figure 3.7) is placed in between the input boxes and the output box. Inside the body box the developer is able to define conditions or to make definitions, which are necessary for the query, i.e. the body box specifies which input events are combined in what way to derive the complex event. The body box is able to use the variables, which are defined in the input boxes of the same query box, and can transfer its results to the output box. A body box consists of an arbitrary number of lines, called body parts. There exist three different types of body parts: conditions, grouping and aggregation, and definitions. In the visual representation each part has its own symbol (compare Figure 3.8). The following text gives a short overview of the three body parts.

- **Condition**: The provided information of this type of a body part corresponds to the where-part of a DURA rule. Like in DURA there exist temporal and data conditions. A temporal condition, for example, defines that two events have to occur within five minutes. A data condition instead may check, if the temperature attribute of a temp event exceeds a given temperature value. Only those events match the query, which fulfil the defined conditions. If only one input is given, this body part may serve as a filter.

- **Grouping and Aggregation**: This body part always consists of two components, namely “Group by” and “Aggregate”. Thereby the user first decides according to which attributes he wants to group the related events (for example by the area) and afterward he defines how to aggregate these groups (for example by the average temperature). In DURA there is the “group by...aggregate” construct, but aggregation

![Figure 3.7: Body box with three body parts: one definition and two conditions](image)
may also be indirectly specified in the head of the rule. The “group by... aggregate” construct in DURA was inspired by the findings of this bachelor thesis with regards to the needs of a visual language.

- Definition: This body part allows to define new variables and can be also placed at any position inside the body box. In DURA the user can declare this in local definitions and the head of the rule. Again the local definitions of the current version of DURA were inspired by the findings of this bachelor thesis.

As mentioned before, a body box may contain an arbitrary number of body parts, which the developer can place in every possible order with respect to the types of the body parts. The reason is that no fixed query pattern is necessary. Of course further constraints, like the set of defined attributes restrict the correct applications of the body part. All in all this aspect of the editor is more flexible than the version of DURA at the beginning of the thesis and inspired equivalent changes in DURA. The body parts are applied bottom-up, which means that variables, which occur in some body part have to be defined in the same or a preceding body part. The user has to keep this fact in mind, when changing positions of the single body parts. Otherwise, alternating the body parts may introduce errors in the program. In Figure 3.9 you can see the previously introduced objects of the visual language, namely event type and query box. The event types temp and avg-temp are the two input event types for the query, represented in the input boxes with identifiers i and j. The event type high-temp is the output event type of the specific query, which is shown in the output box. The body box, consisting of two conditions in this example, is placed in the middle of the query box. The conditions specify which events match the query.

3.4 Stream edges

The last elementary concepts of the visual language for DURA are stream edges. They graphically connect the previously introduced components, event (or state) types and query boxes. Every edge is directed. Together with the event types the edges show the flow of events through the program. Depending on whether the edge goes from an event or state representation to a query box or from a query box to an event type representation, it has a different role. Edges running from an event or state type to a query box are called input edges. Edges running from a query box to an event type representation denote the output of a query and thus are called output edges. The following text gives an overview of the underlying concept in more detail.
Figure 3.9: Query box with its corresponding input events temp and avg-temp, as well as its output event high-temp. Input and output events are represented in the input and output boxes of the query box.
3.4.1 Input edges

The first type of stream edges is the input edge. This directed edge connects an event or state type with an input box inside a query box. Beside the edge, a symbolic reference to an event or state type exists in the head of the input box. A query frequently needs more than one input, thus multiple input edges from one or more event and state types may enter a query box. If there is more than a single input, the corresponding events are combined in a conjunctive manner (compare Section 3.3.1). It is important that although one query box can be the target of multiple input edges, an input box only belongs to exactly one edge and the corresponding event or state type. One event type, of course, is able to be the source of an arbitrary number of input edges even to the same query box. Analogously to the input boxes, there are three types of input edges, namely normal, negated and existential edges (compare Figure 3.10). A negated input edge always corresponds to a negated input box, and so on. With the objective of emphasizing the negation and thereby indicating the negation to the user. It is also possible that one event type occurs in different roles, i.e. for several times, within the same query. In this case several input edges can lead from the unique representation of an event type to an equivalent number of input boxes inside the query box (compare Figure 9). This is necessary, because every event type is only represented once in the diagram.

3.4.2 Output edges

The other type of edges is the output edge, which leads from the output box of a query box to the representation of an event type. This event type is the output event type of the corresponding query box. Therefore it is possible that output edges of multiple query boxes lead to the same event type. In that case, the outputs of these query boxes are combined in a disjunctive manner (compare Section 3.3.2). Unlike input edges, there is only one type of output edge, the normal directed edge.

Figure 3.11 and 3.12 illustrate stream edges and the composition of a single query on the whole. Figure 3.11 shows a query, where the input events are both of the same event type, but represent two different temperature events (compare the example of multiple inputs of one event type from Section 3.4.1). The first temp event is used in a positively way, the second temp event is negated. The negation of the temp event is indicated by the negation symbol on the edge and the input box. The output edge leads from the output box to the derived sensor-broken event type. In Figure 3.12 the running example is accomplished with input and output edges connecting the event types and the query box. This time one event type is linked to the corresponding input box with an existential edge and therefore represents an existential event in the query.
Figure 3.11: Possibility to include two or more events of the same type in one query box. The first temp event is used in a positively way, the second is negated.
Figure 3.12: Adding stream edges to the known example high-temp. The event type temp is connected by a positive edge, the event type avg-temp by an existential one.
4 Visual Editor

After introducing the concepts of the visual language for DURA in the previous chapter, this section describes the main concepts to display the elements of the visual language in a clearly-arranged manner within the corresponding visual editor. The main concepts therefore are zoom, folding and filter. Furthermore, the section explains how the individual concepts can be combined for a dynamic interaction within the editor. But first, the following part of the text gives a short general introduction in the visual editor and its functions. One goal of the visual editor is to display the relations of the individual event types as a kind of data flow diagram, where they proceed from the bottom up. As explained in the previous chapter, some event types derive others, which then can derive new event types themselves, and so on. The visual editor emphasises this aspect by providing a well-arranged data flow structure for its visual components. In Addition to the event type representations, state type representations are also part of the diagram, but they can have many occurrences and do not have a predecessor in the view. Therefore they are not a part of the actual data flow. Query boxes can be folded in to so called “black boxes”. Black boxes representing a query box are displayed using a unique query box symbol. In addition, the attributes of the event and state type representations can be hidden, by folding the corresponding boxes. In this way, details of the query and the types are hidden and the visualisation is more focused on the stream of events. With this data flow perspective, the user has an overview of the whole DURA program. In addition, user-defined perspectives of the diagram are supported. These user-defined perspectives may change, for example in the amount of details specific visual elements are providing for the user. As mentioned before, the visual editor provides three dimensions of interaction, namely zoom, folding and filter. The following text describes them in more detail.

4.1 Zoom

The first concept of the visual editor is the continuous zoom functionality. When the program is zoomed out completely, the data flow structure of the diagram is visible. All zoomed in, one query box with every detail is displayed on the screen. The normal zoom provides the function of scaling up and down the actual screen content, like changing the font size and extending the visual components. The zoom function can be used at any times.

4.2 Folding

The second dimension of the visual editor for DURA is the folding of the visual elements and thereby hiding information from the user. Folding is applied to a number of concepts of the visual language. Folding an event or state type means a hiding of the attributes and their data types within the visual representation. Figure 4.1 shows the representation of the event type temp. The first part of the figure shows an unfolded event type representation, where its attributes are visible. The second part shows a folded type box, therefore the attributes are hidden. The symbol in the right upper corner is a button that the user can click at for folding or unfolding the box. Furthermore, a query box can be collapsed to a black box. In this case the whole content of the query box, namely input, body and output boxes, is hidden and the query box is reduced to a simple symbol. In contrast to collapsing the entire query box, every nested box can be folded and unfolded individually as well. Thus an unfolded input box shows all attributes of the corresponding event type, which are used for an implicit or explicit variable declaration within the query box. A folded input box displays only the event identifier and a horizontal list of the variables.
defined inside. The output box behaves quite the same and presents a list of the variables and the output event name in the headline, when being folded. An unfolded body box shows all body parts and their expressions. When a body box is collapsed, the used body parts are reduced to a list with their corresponding symbols, instead. This folding functionality leads to a reduction of the visible complexity. This is beneficial especially for parts of the program that a developer does not actually work with and therefore might only distract him. That gives him a good overview of the dependencies of the program, without providing too much details at first sight. Additionally, folding effects in a great saving of space on the screen. Section provides further information to the folding concept, especially concerning its underlying interaction concept.

4.3 Filter

The third graphical dimension of the visual editor is the filter functionality, which allows the developer to focus on specific event and state types. For that reason the visual editor provides so called multiquery boxes, which are a “condensed view” for several event and state types, query boxes and stream edges. These multiquery boxes are represented as another kind of “black box” similar to the collapsed query box but with another unique symbol. Multiquery boxes are also integrated in the remaining data flow representation. Figure 4.2 shows such a multiquery box as a connecting element of three input events (smoke, high-temp and sos-call) and an output event (uncertain-alarm). With this perspective the user is provided only with the information, that under specific circumstances the three input types lead to an uncertain-alarm event. The user is not confronted with the exact steps leading from the input events to the output event. In contrast Figure 4.3 shows the same part of the program, when the multiquery box is unfolded. The “black boxes” displayed in Figure 4.3 represent folded query boxes, thus the symbol in this “black box” is different to the symbol of the multiquery box. The use of multiquery boxes achieves a diagram, which provides a better overview of the program to the user, because it hides implementation details and parts of the program, one specific user is not interested in. Filtering also provides the possibility of distinguishing between public and private parts when visualizing a DURA program. Therefore, at first only those elements of the program are fully displayed, which are marked as public. In addition also private parts of the program are shown, when they were marked from one of the developers, actually working on the diagram. Furthermore filtering might be beneficial if a various number of developers is working on the same DURA program, but each of them only wants to see those parts of
the DURA program, which are relevant for his work. The visual editor provides different techniques for defining a personalized view. A personal view means a set of elements of the program, particularly event and state types with query boxes in between, which are relevant for the individual user. The first possibility to get a personalized view is by hiding single event or state types manually from the screen. With the second possibility, the developer can assign keywords, so called categories, to the event types. This approach is quite similar to tagging. An event type can hold an arbitrary number of categories. Thus the user can easily assign already existing categories from a list to other event types, as well. Afterwards, the developer can select his favoured categories and finally obtain a personalized view of the diagram. In this personalized view only those components are visible, which hold one of the before selected categories. The third possibility is to filter information via the selection tool. Therefore the user has to draw a border around a set of visual components and collapse them to a multiquery box.

4.4 Dynamic interaction

After describing the three graphical dimensions of the visual editor individually, this sections deals with a meaningful linking of them. First of all, the editor provides a combination of zoom and folding functionality, which is automatically applied when the zoom function is used. Accordingly, zooming not only enlarges the components on the screen, but also unfolds them at the same time. Thus, the deeper the user zooms in a specific part of the diagram the more information on the zoomed elements is displayed to him. A similar behavior arises, when the user double-clicks on a query box. More precisely, this action beside unfolding the box also zooms to the selected query and shows details of its nested boxes. A double-click on a multiquery box instead, combines the concepts of zoom and filter. This means that the visual editor zooms to the selected part of the program and shows the previously hidden components, when unfolding the multiquery box.
Figure 4.3: Representation of an unfolded multiquery box, where all before hidden elements are displayed. The query boxes are folded and therefore are displayed with their unique “black box” symbol.
5 Interaction

This part of the bachelor thesis deals with different kinds of interaction for the visual editor. These interaction concepts have the aim to increase the ease of use for the developers, while using the editor. Every visual component provides a set of context buttons. Every button is representing a different kind of interaction, which can be performed for the specific visual element. The context buttons of a specific graphical element are only displayed, when the user moves the mouse pointer over the desired element. The following text describes the different interaction fields, like adding and editing, and explains them for every object of the visual language (compare Section 3). This approach is necessary, because the individual components may differ in the way they have to be handled within the editor (this will be getting clearer in the following sections).

5.1 Adding

The add functionality is the first relevant type of interaction for the visual editor. Adding means that the user places a new graphical component on the screen, like an event type representation or a new attribute. This new element then becomes part of the DURA program and can be used to write rules for deriving complex events. For that purpose the visual editor provides different techniques for adding a new component to the program. The following text describes these different kinds of adding-interaction for different types of graphical components. Some of the interactions several components may have in common, others are unique for a specific kind of visual element.

Event type, state type, query box and stream edges  The main parts of the visual language, namely event and state types, query boxes and stream edges, are also the main components of the visual editor and therefore have a lot of interaction in common. One way of adding components to the editor screen is via the toolbar. Figure 5.1 shows a toolbar, which contains small representations of an event and state type, a query box, the three different stream edges and a selection border. This toolbar is part of the visual editor and is placed at one edge of the screen. With the toolbar the user is able to position one of the visual objects on the screen. Therefore he first has to choose one of the desired components from the toolbar and then click on an area of the screen, where he wants to place it. Furthermore, the developer can use the toolbar in a more enhanced way and place the objects on the screen via Drag&Drop. In addition to the toolbar, a context menu is available. This menu appears on the screen when clicking on the right mouse button. The user is able to choose different kinds of interaction from this menu, for example adding a new component. If a new event or state type representation is added, it is necessary to override the default value for the name of the event or state type representation (“EventType” or “StateType”) and to allocate a unique name. This can be achieved by clicking on the specific type (compare Section 5.3). If a state type is added,
the user is shown a list of already existing names of state types by content assist. This is reasonable, because the representation of a specific state type can occur for arbitrary times in the diagram (compare Section 3.2). The corresponding attributes of the existing state type will then automatically be added to its new representation. In contrast to that, the representation of an event type occurs only once in the diagram and therefore needs a unique name when adding the event type to the screen. The actual event type may exist before the adding but just not shown in the diagram. An error occurs, if a new event type is named with an already existing event name.

**Multiquery box** As introduced in Section 4.3 there exists the possibility of creating so called multiquery boxes within the visualisation. These multiquery boxes are “black boxes” with a unique symbol, hiding an arbitrary number of event and state type representations and simple query boxes from the user. Similar to the adding interaction of the components previously described, there are different ways of interaction for creating such a multiquery box. As mentioned in the previous paragraph, the toolbar provides a selection border tool. With this tool the user can draw a border around any part of the diagram and thereby mark the concerned visual components. Then he can click on the selected elements and collapse them to a multiquery box. To make this happen, a context menu appears, where the possibility “Hide selection” can be applied. Furthermore, the visual editor automatically creates multiquery boxes, when the user is working with categories (compare Section 4.3). Therefore he selects the flavoured categories and consequently, only the involved graphical elements will remain on the screen. More precisely, the remaining elements are the event and state types tagged with one of the selected categories, as well as the corresponding query boxes and stream edges between those types. The objects, not matching the categories, do not simply disappear, but are collapsed in a minimum number of multiquery boxes. By this means the user obtains a more personal view of the diagram, where only information relevant for him is displayed. (For a closer look at this concept see Section 4.3.)

**Attribute** Every event and state type may have an arbitrary number of data attributes (compare Section 3). First, when a new event type representation is created, it has no attributes. Thus, there has to exist a way, for adding new attributes to the event type. Therefore every event and state type representation provides a context button for adding a new attribute to the type representation. Figure 5.2 shows two possible alternatives for it. With the first kind of interaction the user can rename an imaginary greyed out last line in the event type representation and thereby add a new attribute to the event type. Afterwards, a new imaginary line will appear underneath the new attribute. The second
approach is quite similar to the first mentioned, except there is a boosted lower margin, where the user can click on and add a new attribute to the event type representation. This possibility of adding a new attribute to the type representation will be only visible when the user moves the mouse pointer over the specific type representation. This handling prevents an information overload for the user and saves space on the screen. In addition, the developer can use the toolbar or the type specific context menu to add a new attribute. The context menu appears, when he clicks with the right mouse button on the type representation. As mentioned in section 3.1 every attribute definition of an event or state type has the appearance ATTRIBUTENAME:TYPE. Thus it is also beneficial that this structure is forced in the visual representation of the types.

Variable Every input box in a query box belongs to an event or state type and represents a specific occurrence of this event or state type in the query. Thus the input box holds the same attributes as the corresponding type. But as described in Section 3.3.1 not every existing attribute of the type is shown in its corresponding input boxes. This has two reasons. First of all, a listing of all attributes inside the input box could require far too much space on the screen. Secondly the user may not be interested in every attribute, but only in a few, which are relevant for the query he is actually working on. Nevertheless, if the user wants to bind a variable to an attribute of the event or state type, he has to add a new attribute field to the input box first. Afterwards he only has to write the self-chosen name of the variable in the new attribute field. Then he is presented a list of permitted attributes via content assist, from which he can select the suitable attribute for the new variable. If he chooses a variable name that already exists in that specific input box it means equality of their attribute values. If the developer uses an attribute via implicit binding in the body or output box (3.3.1) the affected attribute will appear automatically in the corresponding input box (without a variable name). The reason is that the input box shows all attributes of the corresponding type, which are used in the query box.

Body part As described in Section 3.3.3 three different types of body parts exist, namely condition, definition and grouping. One possibility to add a body part is via the context menu. For this purpose the user has to click at the body box. A context menu appears where he can choose the desired body part. The chosen body part will be placed above the already existing body parts, because the parts are applied bottom-up. Afterwards the user is able to switch positions of the body parts (see Section 5.4). Another possibility to add a body part to the body box is via the context buttons of the body box. For that reason the body box additionally provides three special context buttons, each representing one of the body parts. When the user clicks at one of the body part symbols the selected part appears in the query box above the other body parts. These two introduced methods of adding a body part are quite similar to the add interactions of other elements of the visual editor. In addition to the previously introduced methods there exist two ways of adding a body part, which are more advanced. To understand the first advanced approach you have to reconsider the design of the body box. Between the single body parts exist little black areas, which are actually buttons the user can click at. After this click action a menu with three elements appears, containing the specific symbols of all body parts. When the user selects one of the symbols in the menu, the corresponding body part will be placed at the defined position. But the user can also choose more specific characteristics for the selected body part. For that reason the developer has to remain at the symbol of the desired body part in the menu. Consequently, the menu extends to the right and provides more possible specifications for the selected body part. To specify a condition, for example, a set of relational operators appears. The developer can choose one of these
relational operators once again. After that the body part appears in a more specialised
form at the desired position inside the body box. Using the toolbar provides the second
advanced approach of adding a body part and is quite similar to the previously described
one. For that reason the user can select a body part with specific characteristics from the
toolbar and place it via Drag&Drop at the desired position inside the body box. Figure 5.3
shows the fragment of the toolbar, where body parts can be chosen (located in the upper
part of the figure). In this example the developer selects a condition as new body part.
Consequently, the toolbar extends downwards and more possible specifications for the
selected body part are provided. The user can either choose a temporal or data condition.
(Please note: There exists no differentiation of temporal and data conditions on syntax
level. The differentiation is only done for the graphical representation of the condition
parts to support the user in his work flow.) After deciding for a data condition a list of
symbols appears, for example relational operators, where the developer can choose one of
these once again (in this example an equals sign). Now the user can drag the previously
specified body part from the tool bar and drop it on the desired position within the body
box. This graphical representation of a data condition contains two placeholders, for
example for variables, and a relational operator. But not every data condition looks that
way. It is only an example to demonstrate the functionality of body part specification.
The user does not need to select a new part in such detail but could stop, for example, after
selecting the data condition. The tool bar will save the selected specifications of the body
part, until the user chooses new specifications. Additionally to the previously introduced
possibility of adding a condition as new body part to the body box another concept exists,
which has to do with so called macros. Therefore simple conditions like the temporal
condition “before” (event A occurs before event B) are predefined in relations. The user
can utilize these functions either standalone as a simple condition or for the definition of
a macro. Thus, a macro is a sort of composite relation, containing an arbitrary number of
simple relations or even other macros. The goal of this concept is to reduce redundancy,
thus a complex condition occurring arbitrary times has only to be defined once by the user.
Consequently, the use of macros saves space on the screen and supports the developer in his
work flow. As mentioned before, data conditions have a fixed structure which is enforced.
Such a specific structure is given for the other body parts as well. However, it is important
that only reasonable structures are allowed in the body parts. A condition for example

Figure 5.3: Toolbar for the body box, where new body parts can be specified

21
will either require some kind of relational operator or a macro (even if it is not directly specified), otherwise an error occurs. This concept restricts the options of interactions and therefore supports the developers usage of the visual editor.

5.2 Deletion

The second interaction concept of the visual editor is the delete-action. The possibility to delete a graphical component, may differ from one element to another. When dealing with some elements, for example a state type, we have to differentiate between the deletion of a graphical representation on the screen and the deletion of the actual type. Therefore the visual editor provides two special context buttons for every visual component. The first context button, represented by a eraser, deletes the representation of the component from the screen, but still saves the corresponding model element in the program. The other context button is represented by a dustbin and actually deletes the selected component from the program. The following text differentiates between the deletion of the various graphical elements.

General techniques for event, state, stream edge and query box  If the developer wants to delete one of the main components, namely an event or state type, a stream edge or a query box, he can simply drag them into a dustbin, which is located in the toolbar. The deleted objects will be stored in the dustbin, thus the developer can get them out and use them again. This might be reasonable when he has eliminated a element by accident, for example. Another way to delete a component is by using the context menu and then selecting the menu entry “delete”. Moreover, the two context buttons for the delete action are provided for every element, as previously described. Finally, the developer can draw a border around a various number of objects, mark them and delete them all at once. The border tool is provided in the toolbar, as described in Section 5.1. The following text describes the interaction with the different objects and explains possible restrictions for them in more detail.

Event  When the user decides to delete an event type from the diagram, he has to decide whether to delete the event type representation of this diagram or the actual event type. For this reason the two different context buttons for the delete interaction exist. The event type may have several representations in different diagrams of the same DURA program. Or to describe it the other way round, one diagram does not have to contain all in the program specified event types. When deleting an event type from the diagram not only the representation of this event type will disappear, but also its connected stream edges. This is reasonable, because it is not desirable to show edges, leading nowhere. In some cases it might additionally be needed to delete the corresponding input boxes of the specific event type. Otherwise the queries, which refer to the event, cannot be matched and cannot derive a complex event. But possibly the user wants to replace the deleted event type representation with another one and therefore likes to use the same input box and its variable definitions. In this case, the input box should not be deleted. Analogically the visual editor deals with query boxes, where the output box leads to the detected event type. Sometimes the developer might want to change the derived event type from a special query, but in other cases the whole query box might be dispensable and therefore could be deleted, as well. Consequently, it is meaningful to let the developer decide, which alternative he prefers. For this purpose a pop-up window shows up and asks the user to choose his favorite approach.
State One state type may have several representations occurring in the same diagram (compare Section 3.2). Similar to the event type we have to differ between deleting one occurrence (with its edges) and deleting the real state type. The deletion of the state type affects a deletion of all state type representations in the diagram.

Query box Similar to the representations of the event and state types, the deletion of a query box also causes a deletion of the corresponding stream edges.

Stream Edge A stream edge will either be deleted, because the user wants to remove this special edge, or because he eliminates the connected type representation or the query box.

Attribute When talking about the deletion of an attribute in this paragraph, then the attribute definitions in event and state type representations are meant. The attributes in the corresponding input boxes are only views of the actual ones. Therefore, a deletion of these attribute representations in the input boxes do not have consequences for the event and state type representations or the occurrences in other input boxes. Moreover, in most cases the attributes in the input boxes will automatically appear and disappear, without a special interaction of the developer. The appearance of these attributes often depends on their utilization in the rest of the query box. Nevertheless, when the user wants to eliminate an attribute from a event or state type representation he can either use the context buttons or the context menu. In the context menu he has choose the “Delete” entry. If the attribute is not in use in any query box, this method will easily delete it. Otherwise, a pop-up window will appear, asking the user, if he really wants to delete the attribute. Additionally the user has the choice either to to delete all attribute representations in the corresponding query boxes at the same time, or to leave the query boxes as they are. When the user decides not to delete the attribute representations in the query boxes, errors occur, which the developer has to correct himself. The approach, not to delete all attribute representations from the query boxes at once, is reasonable, when thinking of a cooperative work of multiple developers at the same time.

Variable In contrast to attributes variables are local to one specific query box. Thus, a variable can be deleted more easily then an attribute. To delete a variable a x-button is placed next to every explicit variable in the input box, as Figure 5.4 demonstrates. As known from similar concepts, for example the adding of visual elements to the diagram (see Section 5.1), this symbol only appears, when the cursor points to a specific variable in the input box. Alternatively the context buttons can be used to delete the variable. The deletion of a variable will simply be accomplished, without the appearance of a pop-window. If the user deletes the variable by accident, the undo function can be used to fix
it. As mentioned in the paragraph before, the deletion of a variable will not delete the involved attributes, but hide them for the certain input box.

**Body part**  Body parts do also have a x-symbol at the right border or the two context buttons, with whom the user can delete the body part. Furthermore, it is possible to drag a body part to the dustbin or to delete it via the context menu. A restriction for deleting a body part does not exist.

### 5.3 Editing

The interaction for editing different kinds of components within the diagram can be performed in the same way in most cases. Editing means that the user can change the value or name of a specific element, e.g. the name of an event type representation or the content of a body part. Therefore he has to perform a slow double-click on the favoured part and then to enter the new value afterwards. In some cases, for example when dealing with state type representations or data types for attributes, content assist supports the user to find the right value more quickly. When the user wants to name a state type representation, for example, the content assist provides names of already existing state types. This is reasonable, because one state type will have a various number of representations in one DURA program. When binding an attribute to a variable in an input box, content assist will also help the user to find the right attribute name more easily. Therefore it provides only those attribute names, which are defined in the corresponding event or state type representation.

### 5.4 Positioning and movement

Most of the elements in the visual editor do not have a definite position, but can be moved manually by the developer or automatically by the program. The following section describes the underlying interaction concept, first for the top-level elements (event and state type representations, query box and stream edges) and then for the different nested boxes of the query box.

**Visual objects in the graph**  When talking about the visual objects of the diagram this refers to the main components of the visual language (event type, state type, query box and stream edges) (see Section 3). If desired, the user can move them to any possible place within the editor screen. The aim is, to give the developer a personal view of the diagram where he can decide on an optimal arrangement himself. The personal view is only visible for a specific group of developers, thus an arbitrary number of different views can exist for the same project. The various views do not only differ on the arrangement of their objects, but also on the set of objects, which are actually displayed (compare Section 4). The user can also switch between different views. As said above, the arrangement of the personal view can be manually changed by the user. Additionally there might be some kind of tidy-up function, where the components are automatically disposed via a specific algorithm. With this feature the developer has the possibility to automatically arrange the personal view or some specific parts of it. For a local rearrangement the user can select a set of elements from the screen and use the tidy-up function. To reduce confusion during a local rearrangement, the objects will move on the screen slowly, until they reached their final new position. Thus the user has the chance to trace the changes, which helps him to retain orientation later on.
Input box  The input boxes are situated at the bottom of the query box within the so-called input area. In this area the various input boxes are placed next to each other. A new input box automatically appears in the query box, when the user connects one event or state type representation with the query box. It is also possible to add an input box manually to the query. At first the input boxes are arranged in the order of insertion, thus the latest input box is placed rightmost inside the input area. Later on the developer is able to switch their positions, as desired. For that he only has to click on one input box and drag it to his favoured place inside the input area. The other boxes will automatically clear space for it and proceed to their new position.

Body box  The body box is placed in the middle of the query box, between input area and output box. The body parts are arranged as a vertical list and behave quite similar to the input boxes. The developer can move them freely and switch their positions. But he has to be keep in mind that a rearrangement of the body parts may change the meaning of the whole query or even trigger errors. In contrast to the input boxes the user cannot move the body box as a whole.

Output box  The output box is placed at the top of the query box and does not contain any child boxes. Similar to the body box, the user cannot move it, because no need for such an interaction exists.

5.5 Selection

When the user wants to select a visual component from the screen, he usually has to click on the desired element once. In some cases also a click on a button or a double-click causes a selection. This visual editor tries to react as plausible as possible, thus the user can act like he is used to and does not get confused. The following section describes the selection functionality for the single graphical components.

Event and state type  To select an event or state type representation the developer has to click on the favoured type box. After the selection the user is able to perform all other kinds of interaction, like adding an attribute to the type representation or moving the type box to another place on the screen. When the user double-clicks at the type box, the program zooms in on the specific box and unfolds it.

Stream edge  When the user selects a stream edge, he can perform additional interactions, like dragging one end of the stream to another visual element or changing the type of stream edge. When the user wants to change the type of stream edge, for example a negative edge to a existential edge, he has to click twice on it. After that, the edge and the corresponding input box will change their appearance. This is an easy way for the developer to make little changes, without the requirement to delete the old edge first and then create a new one. The order proceeds thereby this way: positive to negative to existential to positive, and so on.

Query box  Similar to the other graphical elements, like type representations, the user can select a query box by clicking on it. Afterwards he is able to perform additional kinds of interaction.
5.5 Selection

Multiquery box To select a multiquery box the user has to click once at its graphical representation (“black box” with a unique symbol). Now he is able to click on the plus-symbol in the right corner of the multiquery box or to double-click on the “black box”. In both cases all involved graphical elements are subsequently visible on the screen and the symbol of the multiquery box disappears. To show which elements actually belong to the open multiquery box, a frame remains around them. Additionally, the plus symbol of the folded multiquery box changes to a minus symbol and moves to the upper-right corner of the frame.

Variable If the developer wants to select a variable, he must click on it once. As shown in Figure 5.5 all occurrences of the selected variable (implicit and explicit) in the query box will be highlighted in the same color. Thus, the user can easily trace the usage of the variable through the query. This may become important, when a huge variety of different attributes and variables are used in one query. Further occurrences of the selected variable, which are inside a folded nested subquery are not visible on the screen. Therefore the affected input box will also be highlighted in the same color as the other variables.

Attribute As described in Section 5.2 attributes in the input and output boxes are only representations of the actual ones, defined in the event or state type representations. Therefore we also have to differentiate between the selection of attributes in the event or state type representations and the selection of attributes, which are used inside the input and output boxes. By clicking on the attributes inside an event or state type representation, the user highlights the attribute in a specific color. Attributes in other type boxes, which are equal to the selected one, will be highlighted in the same color, as well. Thereby it is important to understand that attributes in event and state type representations are not equal, when they share the same name, but when the specification in the query boxes ensures their equality. This may for example be the case, when a temp event holds an attribute “area” and its derived event high-temp owns an attribute “zone”, but it is ensured in the query box that the attributes represent the same area. When the user clicks on an attribute inside an input or output box, all explicit and implicit bounded variables in the corresponding query box will be highlighted. Actually, the behavior is the
same, as clicking on a variable inside the query box.

5.6 Folding

As described in Section 4.2 folding is one of the major concepts of this visual editor. With the folding concept the user is able to choose the information he wants to see in the editor view. The following section explains the various ways of folding interaction for the elements and also describes what information remains, when a specific element is collapsed.

**Event and state type** Every event and state type representation has a collapse/expand button. This might for example be a plus (or minus) symbol, which the user can click at and thereby expand (collapse) the type box. In case of event or state type representations the user can hide (display) the type attributes and their corresponding data types. When the mouse moves over the collapse/expand button and remains there for a while, the program will show the attributes dynamically. Which means, that this mouse-over action is like a normal expand, but does not rearrange other graphical elements in the view. Consequently the displaying of the attributes will end, when the mouse leaves the previous position.

**Query box** A collapsed query box is represented by a “black box” and has a unique symbol. With this symbol, the user can identify the “black box” as single query (see Section 4.3). When the developer clicks at the plus symbol, which is also part of the “black box”, the query box unfolds. Thereby it shows all its components, namely input, body and output box. Thus the next level of detail of the query box is visible on the screen. When the user double-clicks at the “black box”, additional to the query box also its nested boxes unfold. Similar to the previously described folding of event and state type representations, a mouse over action shows the contained information in a dynamic way without changing the actual view. Consequently, all components of the visual editor will remain in their previous position, unlike to an active click, where they have to make room for the opening query box.

**Multiquery box** The symbol of a multiquery box is quite similar to the symbol of the collapsed query box, but nevertheless differs slightly from it. The symbol of the multiquery box also shows a plus symbol in the right corner of the “black box”. At this plus symbol the user can click at or move the mouse pointer over it. When clicking at the button, all included event and state types, query boxes and stream edges will appear on the screen. If there is not enough place left on the screen, the other graphical elements will change their positions. A border surrounds the elements, which belong to the opened multiquery box. Similar to the query box, a long mouse over action at the plus symbol of the “black box” causes a dynamically unfolded multiquery box. This is like a actual unfolding, but a rearrangement of the other graphical elements is not necessary. Thus, the displaying ends when the mouse pointer leaves the plus symbol.

**Input box** The user is also able to fold the input boxes within a query box. He does not have to open or close all given input boxes at once, but can select them separately for folding. For this interaction the boxes have some kind of folding button, e.g. a plus symbol, where the developer can click at. Unlike the folding behaviour of the components described before, folded input boxes do still show some information. More precisely, unfolded input boxes display all used attributes and optional variables in a vertical list. Folded boxes in contrast only show the names of the defined variables in a horizontal list. This is useful,
because the developer still knows which variables he can use in the rest of the query box. Furthermore the developer sees from which input box a variable, used in the body box, originally comes from. If there exist different definitions of one variable the user also recognizes that the variables are equal and that there exists an implicit relation between the corresponding inputs.

**Body box** The body box behaves quite similar to the input box with respect to folding. A body box does also have a expand/collapse button. By clicking the button the user can show or hide the body parts of the body box. An unfolded body box shows a vertical listing of all defined body parts and their content. A folded body box displays a horizontal list of body part symbols, representing the body parts, which are currently defined in the body box. Once again, this approach saves a lot of space on the screen but nevertheless still provides relevant information for the user and the actual query.

**Output box** The output box can also be folded and unfolded. Similar to the other nested boxes of the query box it owns a plus or minus symbol to execute the action. The unfolded output box shows all attributes of the output event and their assigned variables. Unlike the input boxes, all attributes of the derived event type are displayed there. The collapsed output box then provides a horizontal list of all variables, used in the query box. Figures 5.6 and 5.7 show a query box for deriving certain-alarm events. In Figure 5.6 all nested boxes are collapsed and therefore show a plus symbol. In Figure 5.7 the input and output boxes are unfolded and show a minus symbol in the right upper corner. In this example the body box whether provides a plus or minus symbol, because there is no content inside it yet, which could be displayed.

### 5.7 Error indication

The indication of errors is an important feature to support the developer in writing correct programs. The visual editor recognizes some errors while the user is actually developing a program, for example when the developer uses a variable, which has not be declared yet. Other errors will not be recognized before compiletime. For example this kind of error might occur, when the body parts are ordered in a wrong way and no result can be achieved from the query. In the visual editor errors are colored in red and warnings are colored in yellow. If an error occurs, the headline of the affected box shows a specific error symbol, like a red x for example. The visual editor displays warnings different. If a warning occurs and no error is recognized yet in the same box, the headline of the
affected box shows a specific warning symbol, like a yellow exclamation mark. If an error already exists in the same box, no specific warning symbol occurs in the corresponding box., because the is already a error symbol placed in the headline. The affected part of the program passes on errors and warnings up to the level of a type representation or query box. However, warnings serve the purpose to spotlight the user to some noticeable part of the program he should consider carefully. For example warnings appear when a variable is declared, but is not used or when a body part is empty or incomplete. To get an additional description of errors or warnings, the user has to place the mouse pointer on the specific warning symbol. In this case an explanation shows up and even proposes some solution to fix the error in the program whenever possible.

6 Implementation of the visual editor in Java

For this bachelor thesis some functionalities of the visual editor have already been implemented.

For the implementation of visDURA the Eclipse frameworks EMF (http://www.eclipse.org/modeling/emf/) and Graphiti (http://www.eclipse.org/graphiti/) were used. EMF (http://www.eclipse.org/modeling/emf/) is a modelling framework with whom the data model of visDURA was implemented. EMF differs between the meta-model and the actual model. The meta-model is used to build the structure of the model. The actual model instead is an instance of this meta-model. Furthermore, the meta-model consists of two parts, namely the Ecore and the Genmodel model. Both of them provides relevant functionality for the generation of the actual data model. With the Ecore model the user can define the relevant classes and their structure. Therefore the Ecore model provides the Elements: EClass, EAttribute, EReference and EDataType. When the user has defined the single components in the Ecore model, he can generate den Genmodel first. Afterwards he is able to generate the actual model, which generates the model code automatically. [4], [10] The framework Graphiti (http://www.eclipse.org/graphiti/) was used to implement the actual visual editor. Graphiti is a framework, which is actually used to build diagram editors on the basis of a domain model like EMF. Graphiti uses GEF (http://www.eclipse.org/gef/) and its integrated toolkit Draw2D, but hides their technology from the user. With Graphiti it is easy to generate a first version of an graphical editor quickly, because it provides a lot of default implementation and a default design of for the editor. A set of context buttons, consisting of the two delete buttons (one for the deleting the visual representation from the screen, the other two delete also
the corresponding model object), for example, is automatically provided for every visual element. For the implementation of other, not usual functionalities the user needs needs to expend much more effort. The Graphiti framework is quite new (released in July 2010) and therefore provides no fully documentation. Furthermore there exists still a variety of bugs, which the developers of the framework try to fix at the moment. [5] Nevertheless, Graphiti provides a lot of useful functionality for the creation of a visual editor and therefore was used to start the implementation of visDURA.

As mentioned before, not all functionalities of visDURA has been implemented yet. The actual version of the editor provides all necessary components to generate queries for CEP and to develop a whole DURA program, i.e. all elements of the visual language (see Section 3) have already been implemented. Furthermore, most of the interaction functionality (see Section 5), like adding components via toolbar, a content menu or a content button, is available. Although not all alternatives for every interaction functionality, described in Section 5, is actually implemented, the user will be supported in his work flow quite good. Actually, the features of zoom and filter (see Section 4) are not implemented yet. This has to be done in future work.

7 Related Work

For Complex Event Processing and also in a lot of different other areas visual editors already exist. The goal of this section is to demonstrate why none of the editors described in the following and none of their underlying concepts meets the requirements to visualize the CEP language DURA. Additionally the following text compares the editors with the visDURA editor proposed in this bachelor thesis and explains how it fulfils the necessary requirements. The first part of the chapter deals with two visual editors, also developed at the “Programming and Modelling Languages” teaching and research unit at the Ludwig-Maximilians-University of Munich, the second part describes two visual editors for modelling simple business-processes and the final part presents two graphical editors for Complex Event Processing.

7.1 Visual editors of the same research group

The “Programming and Modelling Languages” teaching and research unit at the Ludwig-Maximilians-University of Munich has developed a number of visual editors in the past. Most of them were designed for query languages dealing with XML documents and databases. The CEP language DURA is different to these query languages and therefore has other requirements to a visual editor. The following text describes two visual editors for the querying of XML data, namely visXcerpt and visKWQL.

7.1.1 visXcerpt

VisXcerpt is a visual editor for the query language Xcerpt [11]. Xcerpt is designed for querying and transforming XML documents and databases of semistructured data in the internet. The advantage of Xcerpt in comparison to other query languages is that Xcerpt uses patterns for querying the web. Other languages, like XPath and XQuery, follow an navigational approach, instead. Although Xcerpt is easier to handle than other query languages, it was still too complicated for non-professional users and more suitable for programming experts. The visual editor was developed, to simplify the usage of the query language for non-professional users [2]. XML documents have a hierarchical structure, which is reflected by the pattern-based approach in the language, as well as in the corresponding editor. One Xcerpt program consists of an arbitrary number of rules.
Each rule consists of a construct term in the head and a query term in the body. The visual editor provides static and dynamic aspects, which are both relevant for the actual structure and appearance of the visXcerpt editor itself. As mentioned, patterns reflect the hierarchical structure of an XML document. Thus, the representation of patterns in the visual editor depicts the hierarchical structure of the XML documents using nested rectangular boxes ("tabs") (figure). VisXcerpt also provides some kind of graph structure, which is a relevant aspect of XML. Therefore the editor uses hyperlinks, which allows references between single elements within the editor.

After this short introduction in visXcerpt it gets quickly clear, why this editor is not applicable for the language DURA. The visualisation in visXcerpt focuses on the structure of a XML document, i.e. there exist nested rectangular boxes inside a parent element. The child boxes can hold own child boxes themselves and so on, thus an arbitrary depth is possible, even without subqueries.

VisDURA does not provide this kind of nested structure, because it is not dealing with data from XML documents. Instead, visDURA is an visual editor to write rules for the derivation of complex events from a stream of simple events. Thus, the goal of visDURA is the visualization of the event stream.

7.1.2 visKWQL

Another visual language, which was developed at the "Programming and Modelling Languages" research group is visKWQL [1]. KWQL is a query language for the semantic Wiki KiWi. The goal of a semantic wiki is to receive information easier, by using automated processes and queries. The visKWQL editor is simple enough for beginners but expressive enough for experts, too. Thus the editor provides simple rules up to very powerful ones. The visual structure of the editor, as well as hints, which the user receives while using the editor, makes it easier for non-professional users to learn the language quickly. VisKWQL uses a form-based approach, which means that all elements can be represented as a box, which is able to hold nested child boxes. This structure is adopted from the language where nesting of elements is also provided within the rules.

Similar to the visXcerpt editor, the visual concepts of the visKWQL language are not applicable for the language DURA. Because of the document structure of XML, the editor also provides a nested structure of its graphical elements. For the visualization of DURA this nested structure is not needed, because it wants to visualize the event stream, as mentioned in 7.1.1. However, both editors do have some common aspects. For example, both provide a flat learning curve and their structure is very close to the corresponding language.

All in all, both visual editors introduced in this chapter are very powerful for querying XML data, but not appropriate for the visualization of the CEP language DURA.

7.2 Visual editors for the modelling of simple business applications

There also exist some editors for business applications, which are useful to build the workflow of tasks or decisions. Therefore it is desirable to check, either this kind of visual editors are also able to visualize flow of events, or not. The following text will shortly introduce two of those visual editors.

7.2.1 Visual Rules Modeler

Visual Rules Modeler is a graphical editor, which is part of the "Visual Rules Suite" of Bosch. With the Visual Rules Modeler the user can model simple rules for business applications. With those rules companies are able to come to a reasonable decision and
create a work flow. For the modelling of the rules a lot of visual elements are provided in a palette within the editor. The user can drag these elements into the screen to create his own rules. (figure) In the area of business applications, this might be a good tool to easily define the work flow, dependant on simple decisions. But for a powerful language like DURA, where single queries might become rather complex, this kind of editor is not applicable. [6]

7.2.2 Drools Flow
The graphical editor Drools Flow is quite similar to Visual Drools. Drools Flow also allows to define a work flow, based on simple operations. Every operation, like “join” or “split” has its own symbol in the editor. If such an editor would be used to develop a DURA program, this would lead to a confusing program, with an enormous number of different symbols on the screen. However, Drool is a useful editor for displaying simple work flow operations, but not powerful enough to visualize a complex CEP language like DURA. [7]

7.3 Visual editors for Complex Event Processing
In the last chapter, dealing with already existing visual editors, one editor will be introduced, which is actually developed for Complex Event Processing. The following text gives an short overview of it and demonstrates why visDURA could be better used for modelling CEP rules.

7.3.1 SYBASE
SYBASE is a streaming platform for Complex Event Processing, which also provides a visual editor for the creation of CEP rules [9]. Similar to visDURA it follows the concept of Data flow programming. Therefore it breaks a complex program into a sequence of several simple operations, for example aggregate and filter. Every operation is displayed with an own symbol on the screen and is connected to the other operations via an arrow (figure). There are symbols for different kinds of streams, like source stream and derived stream, and also for the stream operations, which are placed between the source and derived streams. Each element in the editor holds additional properties for the specific stream or operation. To show this additional information the user can click at the symbol and thereby unfolds the element (figure). However, SYBASE is not qualified to develop complex programs, because it is quite hard to generate a complex rule with the simple algebraic operations it provides. Additionally, this behavior leads to a huge amount of simple elements on the screen, one representation for every simple operation. Furthermore, the handling of the editor is not intuitive and the developer needs a long period of vocational adjustment. All in all, visDURA is a more powerful editor for CEP, where the user does not have to program like at a level of simple algebra.
8 Conclusion

The goal of this bachelor thesis was to design a visual editor with whom the user is able to generate the deductive rules of the Complex Event Processing language DURA.

As described in Section 3 the structure of the visual language visDURA is very close to the actual DURA language. On the one hand visDURA is powerful enough to represent complex queries in an effective and meaningful way. On the other hand it can also easily be used by emergency experts, because a lot of structure of the queries is provided, thus they do not have to write that much code. The three dimensions of visDURA, namely zooming, filter and folding (described in Section 4), are meaningful tools to support the individual user and to meet his personal needs while using the editor. Thus, he can gain a personal view, where only a selected set of components is displayed or details from specific visual elements are hidden. The different interaction concepts, as introduced in Section 5, help to increase the ease of use for the developer. For the most interactions the user has different possibilities of performing them, which implies a huge freedom in his work-flow. To recap the advantages of the three working directions, the graphical user interface of visDURA further reduces the complexity of use, compared to the language DURA. It is safe to say that visDURA is a powerful visual editor for generating the deductive rules of CEP, especially when it is used by emergency experts.

As described in Section 6 the first step in implementing visDURA has been done. In the future all previously introduced concepts of the visual editor have to be implemented and translated to the DURA language. Thus, it will become possible to use visDURA parallel to the actual language editor. Furthermore, it is a goal for future work to extend the visual editor in so far, that the user is able to generate the reactive DURA rules with visDURA, as well.
8 CONCLUSION
Inhalt der beigelegten CD
References


