

TU-Darmstadt Fachbereich Informatik

QPE - A GRAPHICAL EDITOR FOR MODELING USING QUEUEING PETRI NETS

Diplomarbeit
von
Christofer Dutz
(dutz@c-ware.de)
März 2006



Betreuer:
Samuel Kounev
Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Datenbanken
und Verteilte Systeme

Prüfer:
Prof. Alejandro P. Buchmann, Ph. D.
Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Datenbanken
und Verteilte Systeme

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, März 2006 Christofer Dutz

i. Table of Contents

i.	Table of Contents	i
ii.	List of figures	iii
iii.	Acknowledgments.....	iv
1	Introduction	1
2	Introduction to QPNs	3
3	XML-based QPN Notation	9
3.1	Describing the Format.....	9
3.1.1	Colors	10
3.1.2	Places	11
3.1.3	Transitions	12
3.1.4	Connections	13
3.1.5	Meta-attributes	14
3.2	Schema definition	14
4	Approaches to Java GUI Development	16
4.1	SWING and AWT based Approach	16
4.2	Extending existing solutions: PIPE	16
4.3	Frameworks	18
4.3.1	Touchgraph based Approach	18
4.3.2	Eclipse: RCP, SWT and GEF based Approach	19
5	QPE Tool Design & Architecture	22
5.1	Plugin.xml	22
5.2	Main application (de.tud.cs.qpe)	23
5.3	Rich Content Platform (de.tud.cs.qpe.rcp)	23
5.3.1	Perspective	24
5.3.2	WorkbenchWindowAdvisor	25
5.3.3	Actions (de.tud.cs.qpe.rcp.actions)	25
5.3.4	Menus and Toolbars	26
5.4	Model (de.tud.cs.qpe.model)	26
5.5	Editors (de.tud.cs.qpe.editors)	31
5.5.1	Gef (de.tud.cs.qpe.editors.???gef)	32
5.5.1.1	The palette (de.tud.cs.qpe.editors.???gef.palette)	33
5.5.1.2	The property view (de.tud.cs.qpe.editors.???gef.property)	34
5.5.2	Controller (de.tud.cs.qpe.editors.???controller)	36
5.5.3	View (de.tud.cs.qpe.editors.???view)	39
5.5.4	Net editor (de.tud.cs.qpe.editors.net)	39
5.5.4.1	MultipageEditor	39
5.5.4.2	NetLayoutEditPolicy	41
5.5.5	Incidence function editor (de.tud.cs.qpe.editors.incidence)	41
5.5.5.1	Model (de.tud.cs.qpe.editors.incidence.model)	41
5.5.5.2	Model (de.tud.cs.qpe.editors.incidence.view.layout)	42
5.5.5.3	View (de.tud.cs.qpe.editors.incidence.view)	43
5.5.5.4	View (de.tud.cs.qpe.editors.incidence.view.anchor)	43
5.6	Rule Manager	44
5.7	Problems	45

5.7.1	Unwanted menu and toolbar entries	45
5.7.2	Menu entries that can be checked	45
5.7.3	Adding elements to context menus	46
5.7.4	Removing Event Listeners from Removed Nodes	47
5.8	Integration with SimQPN	47
6	Extending QPE	49
6.1	Some general information about Tables and TableViews	49
6.2	Some general information about Trees and TreeViews	52
6.3	Extending Menu, Toolbar and Context Menus	53
6.4	Adding additional net elements	54
6.5	Adding or changing Nodes attributes	54
6.6	Extending the introduction and help system	55
6.7	Extending or changing the SimQPN wizard	55
7	QPE Tool User's Guide	56
7.1.1	Outline View	56
7.1.2	Property View	56
7.1.3	Console View	57
7.1.4	Problem View	57
7.2	Editing Nets	57
7.2.1	Defining Colors	58
7.2.2	Defining Nets	58
7.3	Editing Incidence functions	64
7.4	Running SimQPN	65
7.4.1	Select Run Configuration	65
7.4.2	Select Analysis Method	66
7.4.3	Simulation Run Configuration	67
8	Conclusion	70
9	Future work	71
10	Bibliography	70
11	Index	73

ii. List of figures

Image 1 Net Editors in HiQPN (left) and QPE (right)	8
Image 2 Incidence function editors in HiQPN (left) and QPE (right)	8
Image 3 Root of a QPE document	10
Image 4 Colors	10
Image 5 Places	11
Image 6 Transitions	13
Image 7 Connections	13
Image 8 Meta-attributes	14
Image 9 Distribution function	15
Image 10 PIPE-based QPE prototype	17
Image 11 Touchgraph application (TwoMore)	18
Image 12 Graphical elements of an Eclipse application	24
Image 13 Access paths using DocumentManager	27
Image 14 Registering a new document	28
Image 15 Registering as change listener	29
Image 16 Modifying a document	30
Image 17 Standard Eclipse property view	34
Image 18 Custom QPE property view	35
Image 19 EditPart structures	37
Image 20 Selected net editor element	39
Image 21 Wrapped element references	42
Image 22 Figure of place with 5 color-refs	43
Image 23 Application overview	56
Image 24 Color editor	58
Image 25 Net editor	59
Image 26 Net prior to paste operation	61
Image 27 Net after pasting to same document	62
Image 28 Net after pasting to new document	63
Image 29 Colors after pasting to new document	63
Image 30 Incidence function editor	64
Image 31 Configuration page	65
Image 32 Analysis method page	67
Image 33 Simulation run configuration page	68
Image 34 Place configuration page	69

iii. Acknowledgments

Most of all I want to thank my parents for the great support during the entire time of my studies. I also wish to express sincere appreciation to Professor A. Buchmann and to Dr. Samuel Kounev for his assistance in the preparation of this manuscript. In addition, special thanks to the countless number of developers active in the Eclipse-Newsgroups for providing us with important information during the implementation-phase of the QPE Editor.

1 Introduction

Petri nets are a special notation for building models of complex systems such as distributed applications or manufacturing systems. Over time several types of Petri nets have evolved, each with its individual enhancements suiting individual needs.

Queueing Petri nets are a powerful formalism that can be exploited for modeling distributed systems and analyzing their performance and scalability. Both hardware and software aspects can be integrated into the same model and is especially well suited for modeling distributed applications. However, currently available tools for modeling and analysis using queueing Petri nets are very limited in terms of the scalability of the analysis algorithms they provide.

Mr. Samuel Kounev addressed this problem by developing a simulator called SimQPN circumventing the state space explosion problems other approaches bring with them. This makes it possible analyzing far more complex nets. The nets this simulator operates on are currently programmed directly into the simulator source-code. Whenever the net is changed, modifying the code and recompiling the simulator is required. In addition to this relatively complicated procedure it sometimes is an error prone task creating the nets, since for developing a net a large set of multidimensional arrays has to be initialized and array-index errors are quite common and mostly lead to errors when performing a simulation run.

This is one of the primary problems addressed by this thesis. By offering a graphical user interface for modeling QPNs and integrating the simulator into this, the development speed and quality of the results would dramatically increase.

Since all types of Petri nets have a graphical notation, special tools are needed in order to construct, display, store and operate on these. Petri nets can have several characteristics and therefore may need different notations and editors.

At first multiple existing tools were evaluated. The 8 most promising tools were: HiQPN (University of Dortmund), PIPE (Imperial College London), Project PED (Vienna University of Technology), JARP Petri Nets Analyzer (Sourceforge), PNS (Michael Ancutici and Thomas Bräunl), Peneca Chromos (TU-Ulmenau), Renew (University of Hamburg) and Design/CPN (University of Aarhus).

The only tool besides HiQPN supporting colored Petri nets was the tool called Design/CPN but unfortunately this tool is a commercial product with no freely available code-base. Apart from HiQPN, the only tool reported to be able to deal with hierarchical Petri nets is a tool called Snoopy developed by the University of Cottbus. Unfortunately the web pages of this project have not been available throughout the entire evaluation phase and no further information could be found.

Since HiQPN allowed modeling exactly the types of nets needed for working with SimQPN, this was the first tool to have a closer look at. Unfortunately this tool is implemented in C and is only able to run on the Solaris platform. SimQPN is written in pure Java, linking the two would

have been a problematic task. On the other side the restriction to Solaris prevents any wide spread usage of the toolset. The requirement to make the suite available on all major software platforms demanded a different approach.

Nevertheless this tool was a great inspiration towards how the editors are built up and especially how the incidence function editor visualization is done.

There were two possibilities: extending an existing solution or creating a new application from scratch.

Since the HiQPN code was written for Solaris using it would have meant porting it to different platforms it was therefore not taken into consideration. One of the most promising tools besides HiQPN was the PIPE project. Work was therefore started on extending PIPE with additional net elements such as queueing and subnet places as well as timed transitions. Then PIPE was extended to be able to configure the relatively complex set of attributes needed for configuring all elements in a colored version since this made configuring colors for places and modes for transitions necessary. Unfortunately the deeper layers of PIPEs architecture prevented adding the incidence function editor. This was needed for modeling the colored transitions incidence functions. Adapting these layers was not possible without using big architectural hacks or refactoring the entire platform.

This is why after trying to extend existing products focus was switched on implementing a new solution from scratch. For this several frameworks were evaluated. Finally it was decided to implement the editor using Eclipse and its Graphical Editing Framework GEF.

In the course of developing a stand-alone Eclipse application with complex graphical editors it became clear that the power offered by the Eclipse platform comes with a relatively high price. Several problems had to be solved; problems that sometimes took several days or weeks for solving. Even receiving a lot of help from the Eclipse newsgroups.

Apart from creating an editor, a data format had to be developed to store the models in XML format. This format should be easily read and understood by humans and still be flexible enough to be extended to future needs without corrupting the old format. Here it was possible again extending an existing format or developing an entirely new one. The only format considered extending was the PNML format developed at the Humboldt Universität zu Berlin which represents a quasi standard in defining Petri nets. Unfortunately this format didn't allow all the extensions needed. So a new format was developed while keeping in mind that importing and exporting (even if not entirely lossless) had to still be possible using XSL transformations.

In the end the results of this thesis were used as a basis for creating a paper which was submitted for publication at the International Conference on Quantitative Evaluation of SysTems (QEST).

2 Introduction to QPNs

The following introduction to QPNs is based upon Samuel Kounev's work: "A Methodology for Performance Modeling of Distributed Component-Based Systems using Queueing Petri Nets", "Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets" and "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation"

In order to fully understand this thesis some basic understanding of queueing Petri net models is required. This chapter intends to provide the needed background. There are a number of different Petri net types. Queueing Petri nets are one of these types. In contrast to conventional Petri net types they allow to integrate both hardware and software aspects into one consistent model. Because of this they are especially well suitable for modeling and analyzing distributed systems such as complex e-commerce systems.

In this chapter the mathematical definition of basic Petri nets will be described and starting from here, the definition of queueing Petri nets will build up by showing how the Petri nets evolved from the basic net definition to QPNs.

Definition 1: An ordinary Petri Net (PN) is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ where:

- 1) $P = \{p_1, p_2, \dots, p_n\}$ is a finite and non-empty set of places,
- 2) $T = \{t_1, t_2, \dots, t_m\}$ is a finite and non-empty set of transitions, $P \cap T = \emptyset$,
- 3) $I^-, I^+ : P \times T \rightarrow N_0$ are called backward and forward incidence functions, respectively,
- 4) $M_0 : P \rightarrow N_0$ is called initial marking and specifies the initial number of tokens inside the place.

If $I^-(p, t) > 0$, then the place p is called an input place for the transition t . If $I^+(p, t) > 0$, then p is a so-called output-place for t . A place can be both input and output place to a transition at the same time. In a graphical representation I^- and I^+ assign natural numbers to arcs these numbers are called weights.

When all input places contain at least as many tokens as the weights of the connections connecting them to the current transition state, it can be thought of as enabled. That transition is then able to fire. Firing can be understood as destroying the number of tokens defined by the incoming connections weights in the input places and creating new ones in the output places.

As mentioned above there have been multiple extensions to this definition of Petri nets. The one being discussed in the next part is the colored Petri net (CPN), which was developed by K. Jensen. These nets allow a type (color) to be attached to the nets tokens. A color function C

assigns a set of colors to each place. By this it defines which types of tokens a place can handle. In addition to adding these new token types, CPNs now allow transitions to fire in different modes. These modes (transition colors) are also assigned by the color function C .

A formal definition of a CPN follows:

Definition 2: A Colored PN (CPN) is a 6-tuple $CPN = (P, T, C, I^-, I^+, M_0)$ where:

- 1) $P = \{p_1, p_2, \dots, p_n\}$ is a finite and non-empty set of places,
- 2) $T = \{t_1, t_2, \dots, t_m\}$ is a finite and non-empty set of transitions, $P \cap T = \emptyset$,
- 3) C is a color function that assigns a finite and non-empty set of colors to each place and a finite and non-empty set of modes to each transition.
- 4) I^- and I^+ are the backward and forward incidence functions defined on $P \times T$, such that $I^-(p, t), I^+(p, t) \in [C(t) \rightarrow C(p)_{MS}]$, $\forall (p, t) \in P \times T$ ¹.
- 5) M_0 is a function defined on P describing the initial marking such that $M_0(p) \in C(p)_{MS}$.

Other extensions allow temporal (timing) aspects into the net definition. For example stochastic Petri nets (SPN) attach an exponentially distributed firing delay to each transition and with this controls the time an enabled transition waits before being fired. Generalizes stochastic Petri nets (GSPN) allow two types of transitions:

- Immediate transitions, which fire without any delay.
- Timed transitions, which fire after waiting a randomly exponentially distributed time.

If more than one transition is enabled at the same time, the firing weight decides which is fired first. Firing of immediate transitions always has priority over timed transitions.

A formal definition of a GSPN follows:

Definition 3: A Generalized SPN (GSPN) is a 4-tuple $GSPN = (PN, T_1, T_2, W)$ where:

- 1) $PN = (P, T, I^-, I^+, M_0)$ is the underlying ordinary PN,
- 2) $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,
- 3) $T_2 \subseteq T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset, T_1 \cup T_2 = T$,

¹ The subscript MS denotes multisets. $C(p)_{MS}$ denotes the set of all finite multisets of $C(p)$.

4) $W = (w_1, \dots, w_{|T|})$ is an array whose entry $w_i \in \mathbb{R}^+$ is a rate of a negative exponential distribution specifying the firing delay, if $t_i \in T_1$ or is a firing weight specifying the relative firing frequency, if $t_i \in T_2$.

Combining definitions 2 and 3, leads to Colored Generalized Stochastic Petri nets (CGSPN):

Definition 4: A Colored GSPN (CGSPN) is a 4-tuple $CGSPN = (CPN, T_1, T_2, W)$ where:

- 1) $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying CPN,
- 2) $T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$,
- 3) $T_2 \subseteq T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset, T_1 \cup T_2 = T$,
- 4) $W = (w_1, \dots, w_{|T|})$ is an array with $w_i \in [C(t_i) \times \mathbb{R}^+]$ such that $\forall c \in C(t_i): w_i(c) \in \mathbb{R}^+$ is a rate of a negative exponential distribution specifying the firing delay due to color c , if $t_i \in T_1$ or is a firing weight specifying the relative firing frequency due to c , if $t_i \in T_2$.

Even if CGSPNs offer quite some possibilities for building complex models, they fail to provide any modeling support of queueing disciplines. This lack lead to the emergence of queueing Petri nets (QPN). In these nets queues and timing aspects were added to the definition of places of CGSPNs. There are two types of places now:

- Queueing places: These possess an integrated queue. Also called timed queueing Places
- Ordinary places: Also called immediate queueing places..

Timed queueing places are separated into two parts: The queue and the depository. Incoming tokens are initially stored in the places queue and after passing the queue they are immediately moved to the depository from which they can be fetched by the incidence functions of transitions. Immediate queueing places can be seen as places with queues that serve incoming tokens immediately. Scheduling in these places has priority over that of timed queueing places. In which order tokens pass from the queue into the depository is defined by the places scheduling strategy.

A formal definition of a QPN follows:

Definition 5: A Queueing Petri net (QPN) is an 8-tuple $QPN = (P, T, C, I^-, I^+, M^0, Q, W)$ where:

- 1) $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying Colored PN
- 2) $Q = (Q_1, Q_2, (q_1, \dots, q_{|P|}))$ where

- $Q_1 \subseteq P$ is the set of timed queueing places,
- $Q_2 \subseteq P$ is the set of immediate queueing places, $Q_1 \cap Q_2 = \emptyset$ and
- q_i denotes the description of a queue taking all colors of $C(p_i)$ into consideration, if p_i is a queueing place or equals the keyword 'null', if p_i is an ordinary place..

3) $W = (W_1, W_2, (w_1, \dots, w_{|T|}))$ where

- $W \subseteq T$ is the set of timed transitions,
- $W_2 \subseteq T$ is the set of immediate transitions, $W_1 \cap W_2 = \emptyset, W_1 \cup W_2 = T$ and
- $w_i \in [C(t_i) \times \mathbb{R}^+]$ such that $\forall c \in C(t_i): w_i(c) \in \mathbb{R}^+$ is interpreted as a rate of a negative exponential distribution specifying the firing delay due to color c , if $t_i \in W_1$ or a firing weight specifying the relative firing frequency due to color c , if $t_i \in W_2$.

One of the biggest problems when dealing with the analysis of QPNs is the fact that most analysis techniques available are based on Markov Chains and are therefore susceptible to the state-space explosion problem. This means that as soon the number of places and tokens of a QPN grows, the size of the state space of the underlying Markov Chain grows exponentially. This limits the size of the net dramatically.

Hierarchically combined Queueing Petri nets (HQPNet) are obtained by a natural generalization of the original QPN formalism. In HQPNs a queueing place may contain a whole QPN instead of a single queue. Such a place is called a subnet place. A subnet place might contain an ordinary QPN or again a HQPN allowing multiple levels of nesting. Every subnet of a HQPN has a dedicated input and output place, which are ordinary places of a colored Petri net. Tokens being inserted into a subnet place after a transition firing are added to the input place of the corresponding HQPN subnet. The semantics of the output place of a subnet place is similar to the semantics of the depository of a queueing place: tokens in the output place are available for the output transitions of the subnet place. Tokens contained in all other places of the HQPN subnet are not available for the output transitions of the subnet place. Every HQPN subnet also contains actual population place, which is used to keep track of the total number of tokens fired into the subnet place.

There are quite some tools available for editing and analyzing Petri nets, but only one tool can handle HQPNs. This tool is called HiQPN unfortunately it is only available for the Solaris platform and all analysis methods suffer the problem described later on in this chapter.

After this there is only one tool available for editing colored Petri nets this is called Design/CPN but is a commercial product with no support for QPNs. Since being a commercial tool, no source code is available.

One tool reported to be able to build hierarchical Petri nets is called Snoopy. Unfortunately this project seems to no longer exist since all references to it lead to dead pages.

The rest of the editing tools that were evaluated all aimed at simple Petri nets. These were in particular:

- PIPE (Imperial College London),
- Project PED (Vienna University of Technology),
- JARP Petri Nets Analyzer (Sourceforge),
- PNS (Michael Ancutici and Thomas Bräunl),
- Peneca Chromos (TU-Ulmenau),
- Renew (University of Hamburg) and
- Design/CPN (University of Aarhus).

Most of them come with analysis methods, but these are limited to the types of nets the editors support, so there are no means for simulating QPNs.

As mentioned in chapter 1 QPNs allow to model both software and hardware behavior into the same model. Currently available analysis tools all suffer the state space explosion problem dramatically limiting the size of nets which modern computers can analyze. The SimQPN simulator by Samuel Kounev uses an alternative approach by means of discrete event simulation.

The following images show the visual representation of QPNs and transitions incidence functions in both the HiQPN and QPE tool. The general presentation is quite similar as the HiQPN tool greatly inspired the way the incidence functions editor visualizes a transitions incidence function.

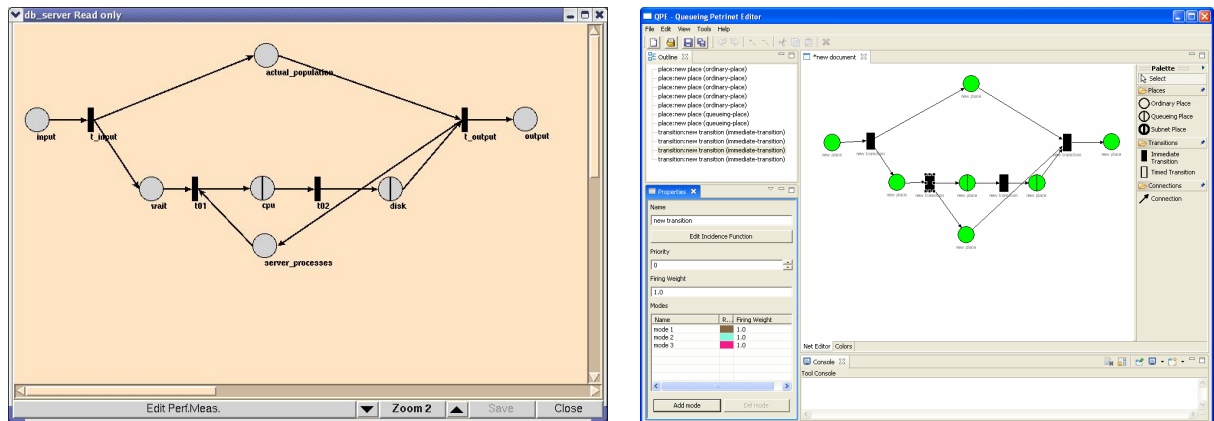


Image 1 Net Editors in HiQPN (left) and QPE (right)

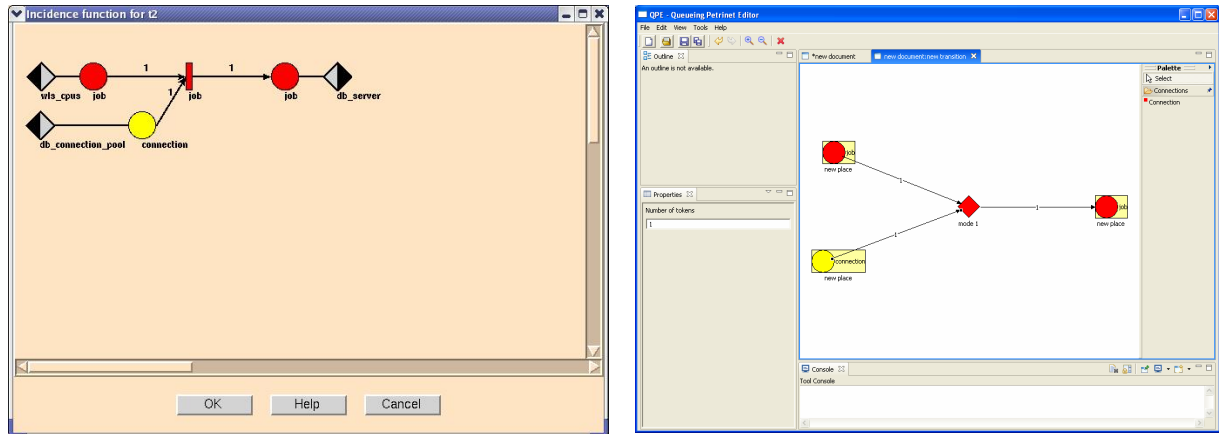


Image 2 Incidence function editors in HiQPN (left) and QPE (right)

3 XML-based QPN Notation

Based upon the definition of queueing Petri nets in chapter 1 the first part of this thesis was to develop an XML-based storage format for queueing Petri nets. Evaluating existing formats was begun, especially the Petri Net Markup Language (PNML) format. At first work was started using a PNML-like format. PNML is intended to be used as universal format for storing Petri nets in XML format. It provides a lot of attributes and extensions that would not be needed. On the one side it prevents us from adding extensions that don't belong to ordinary net definitions.

The main differences between the QPE and the PNML format:

- At first the net-wide definition of colors would not have been possible with PNML
- PNML offers a lot of standard elements, which are not needed
- Mixing places transitions and connections inside the net definition makes reading the XML document difficult

As being developed as an internal storage format my focus lay on the usability in the QPE application. Using an alternate format does not at all prevent us from importing and exporting PNML, because this can be done using XSL transformations without big problems.

3.1 Describing the Format

While developing the editor it became obvious, that the definition of colors on a per place basis is easier to implement, however in practice the same colors are often used in multiple places and therefore they have to be redefined multiple times. This motivated us to define colors on a per-net instead of a per place basis. The net-element contains a colors-element which stores the color definitions. This is why the place elements have no color child elements but color-refs instead. These are simply references to the globally defined colors containing the id of the color referenced. Each of these color-refs possesses a lot of attributes, which ones these are however depends on the place's type. "color-ref"-elements are encapsulated inside a color-refs element.

As it is impossible to predict which additional attributes future tools will need. This is why a mechanism for providing additional information is needed. In QPE this is done using so-called meta-attributes. These can be added to the different levels of the document. Meta-attributes added to the net affect the entire net. Ones added to places or transitions affect only these. Momentarily the lowest level for adding meta-attributes is adding them to modes or color-refs. Even if running QPE without any plug-in or other extension, meta-attributes are used. Since the graphical location of a place or transition inside the editor is tool-specific information and has nothing to do with the actual model, it is represented by a meta-attribute.

Every meta-attribute has one mandatory “name” attribute. This is used to identify the extension this meta-attribute is used for. In the QPE application for example there are “location” meta-attributes for describing the location of an element, “sim-qpn” meta-attributes for storing simulator settings. More than one meta-attribute with the same name can be located in the same meta-attribute container. This is done for the simulator plug-ins configurations. Here a combination of “name” and “configuration-name” used for identifying a single meta-attribute.

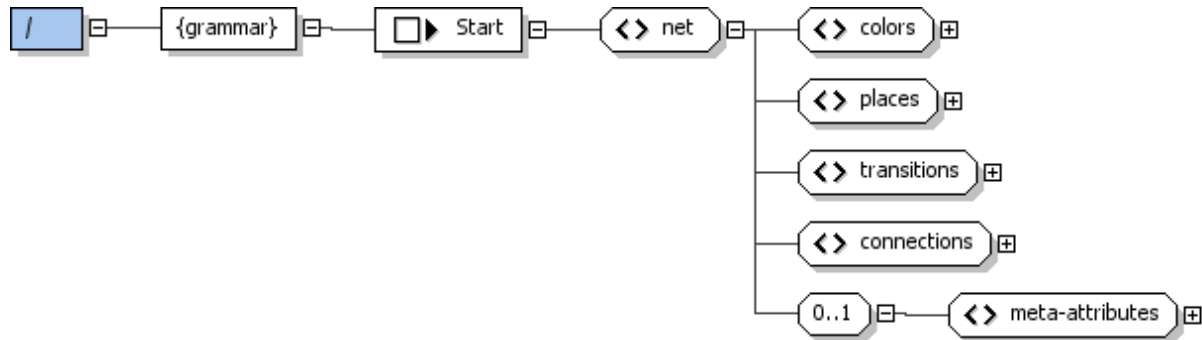


Image 3 Root of a QPE document

In general a QPE document has a “net” root-element. This contains “colors”, “places”, “transitions”, “connections” and a “meta-attributes” element as children. They all serve as containers for the elements. Now to the actual element definitions:

3.1.1 Colors

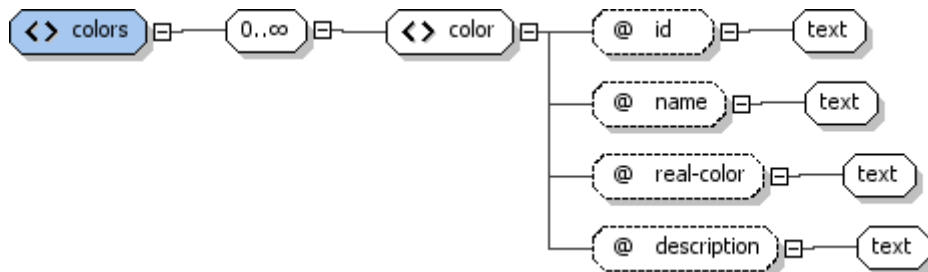


Image 4 Colors

Color elements possess four attributes:

- An id,
- a name,
- a real color, giving the color an RGB color and
- a description giving the user some more information about the usage of the color.

3.1.2 Places

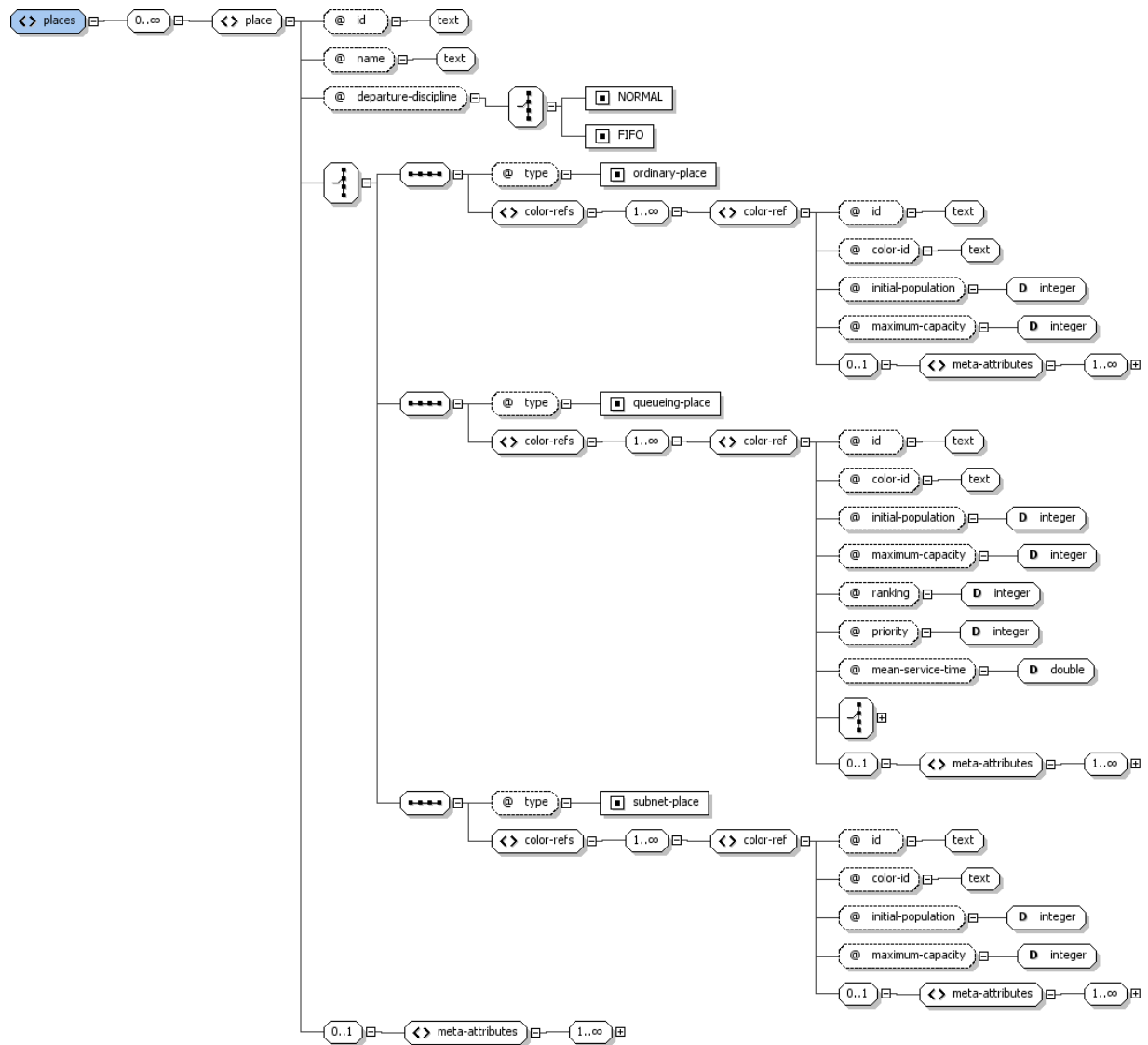


Image 5 Places

All places have the attributes id, name, departure-discipline and type. The only values allowed as departure-discipline are: NORMAL and FIFO. All have color-refs and meta-attributes children. The color-ref child elements of the color-refs element have different sets of attributes depending on the type of the place.

One thing all have in common though is they all have “id”, “color-id”, “initial-population” and “maximum-capacity” attributes. The id is used as source or target in incidence functions connections and the color-id references a color which is defined in the colors-section of the net. Initial-population defines how many tokens are set in the place initially and maximum-capacity defines how many tokens of the given color at maximum the place is able to store.

If the type of place is “queueing-place” then some additional attributes are needed:

- Ranking
- Priority
- Mean-service-time
- Distribution-function and depending on its value up to 3 individual parameters configuring the distribution function.

At the moment, subnet-places have the same definition as ordinary places. As described in chapter 1 they should rather be defined to contain a nested QPN. This will be done later when the subnet-editor is implemented.

3.1.3 Transitions

All transition elements – as places – have id, name and type attributes. The id is used for being referenced in the nets connections.

Depending on the type attribute the mode definitions differ a little. All mode elements have in common the id name and real-color attributes. The only difference is that immediate transitions have a firing weight and timed-transitions a mean-firing-delay attribute.

The incidence function is defined by a “connections” element which has “connection” child elements. These connection elements all have four attributes:

- Id (currently not used),
- Source-id: specifying the id of the source color-ref or mode,
- Target-id: specifying the id of the target color-ref or mode and
- Weight: specifying how many tokens of a given color are needed to activate the connection.

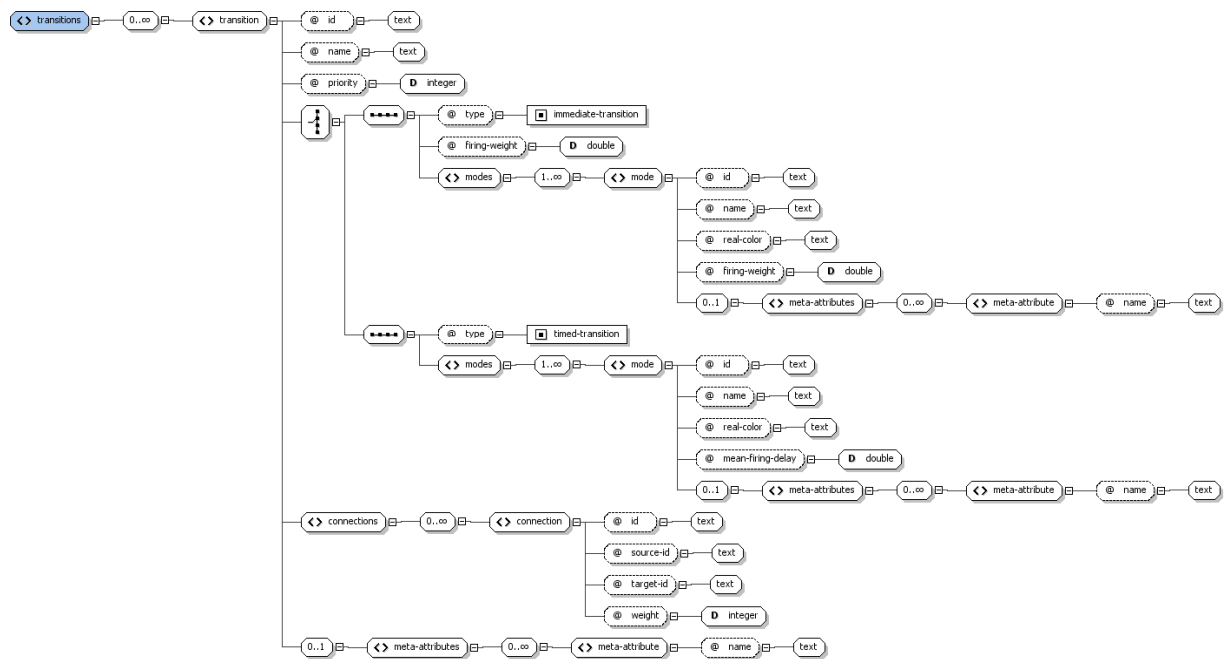


Image 6 Transitions

3.1.4 Connections



Image 7 Connections

According to the definition of Petri nets these connections actually are not part of the definition. They were added to make editing the nets easier. Usually all color-refs of all places would be available in the incidence functions as input and output color-refs, but since this would make dealing with these incidence-functions almost impossible in larger nets, these connections tell the incidence-function editor which color-refs to make available as input and as output color-refs.

Connections are fairly simple elements as they only possess three attributes:

- Id: which (currently not used),
- Source-id, which is used for defining the id of the source place or transition.
- Target-id, which is used for referencing the id of the target place or transition.

3.1.5 Meta-attributes

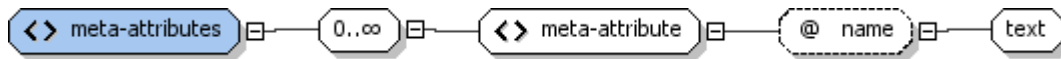


Image 8 Meta-attributes

Most elements in QPE are allowed to have meta-attributes and there are almost no restrictions to the attributes they provide. The only attribute that has to be provided is the name attribute.

3.2 Schema definition

There are several different ways to specify the schema of XML-based markup languages. Each method has its pros and cons. At first the QPE application was developed using an XML format without any explicit document type definition for validation, since it was initially planned as internal data-format. When it came to defining a schema several decisions had to be made.

One problem was to decide the way the different types of transitions and places were described. The problem here was that the different types of places and transitions required different sets of attributes and child nodes. For example the color-references of a queueing place are quite different from the ones of an Ordinary-Place. At first using a standard W3C Xml-Schema (XSD) sounded reasonable, since it is widely used and there are a great number of tools for defining the schemas and for validation.

While developing the language as XSD it occurred that these types of schemas had one big drawback:

In order to make the format extensible to further types of places and transitions, these had to use standard names. Otherwise major refactoring would be necessary. Unfortunately using XSDs it is impossible to define multiple elements with the same name that have different definitions.

In the case of the QPE data format a type of element should not be described by the tag name alone, but by a combination of tag name and a type-attribute. One alternative would have been to use “ordinary-place”, “queueing-place” and “subnet-place” elements instead. The downside of this way would have been that the extensibility of the format would have been compromised. All XPath expressions in the QPE application operate on standard tag names (“place” for places and “transition” for transitions) using individual tag-names would have meant the need of refactoring the entire application for each new node type, or rather complex selection-strategies. After some time of searching for alternatives the RelaxNG format seemed fitting. Since this allows everything needed for the QPE format and there is a reasonable number of editing and validating tools available, this was used as format for defining the main QPE data format. As mentioned above both the fact of adding colors as new types and the usage of a non-PNML language-base does not restrict us from importing and exporting PNML using a simple XSL transformation.

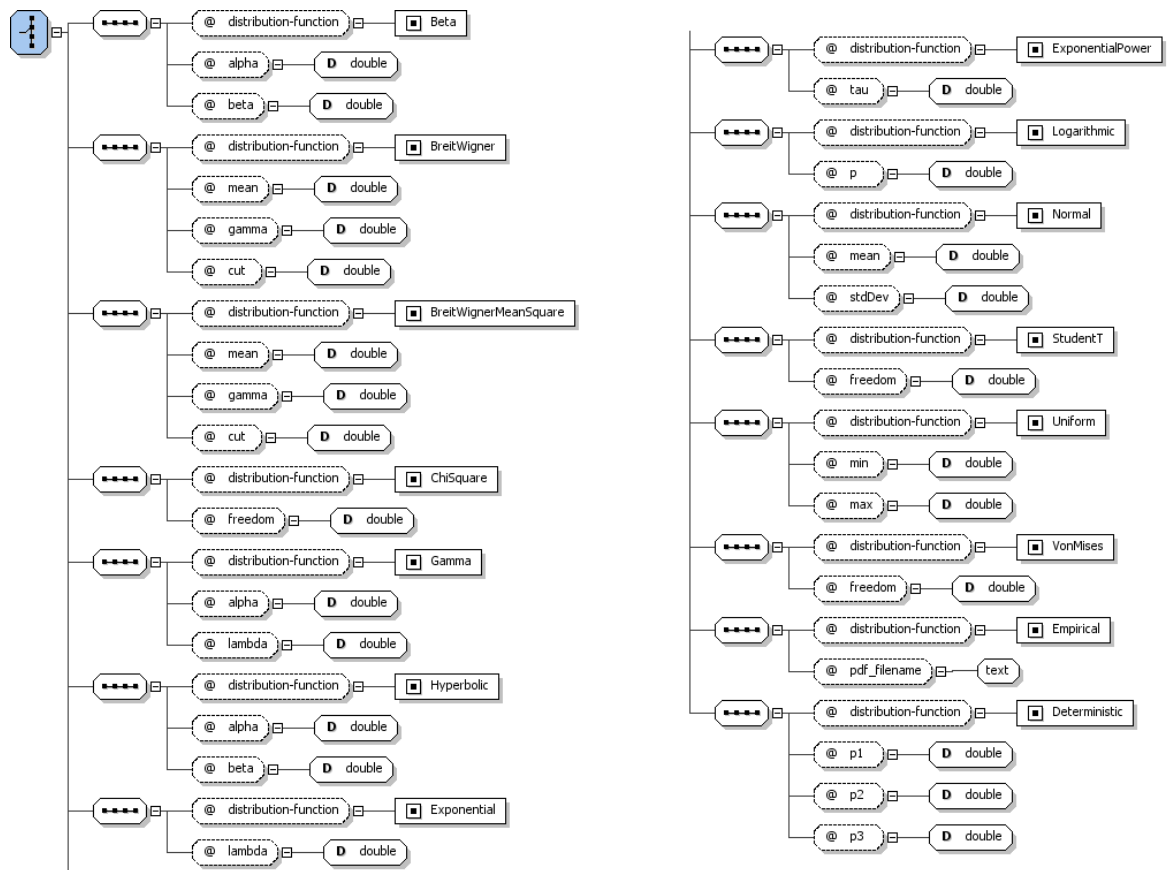


Image 9 Distribution function

4 Approaches to Java GUI Development

In the early days of Java there was no alternative. GUIs had to be built by hand using AWT and later on SWING. In the meanwhile, several frameworks have evolved, each framework having its strengths and weaknesses. In the course of this thesis, both the direct approach and several frameworks were evaluated. The following chapters are ordered in chronological order and show the advantages and disadvantages of the different alternative approaches that were considered using.

4.1 SWING and AWT based Approach

The first approach was the direct one. Since having had fairly good knowledge in developing Java GUIs using SWING and AWT this was chosen.

After first successes in creating the editors for colors and their properties the next task was to create dynamically moveable labeled objects, which were able to be connected with each other using connection objects – the editor-objects themselves. This is where the entire solution started getting very difficult. Not only would it be necessary to create some sort of runtime for displaying our graphical objects, but also syncing this with the model would be a very challenging task.

On the other side reinventing the wheel could not be the solution, so evaluating existing solutions was started.

4.2 Extending existing solutions: PIPE

One quite promising project seemed to be the PIPE Project, which is hosted as open-source project on Sourceforge. PIPE focuses on analyzing ordinary Petri nets. The editor it provides is therefore limited to ordinary places and immediate transitions. The visualization of places is focused on displaying non-colored nets. In order to not only build models PIPE provides a plug-in mechanism for introducing components for net-analysis.

Nevertheless the general visualization engine looked as if it would suit my needs. Adding additional element types and properties should be possible.

After having a first glance at the code quality looked promising, so the next step was to modify the latest PIPE-version to support the additional functionality.

Since the original PIPE version only supported ordinary places, timed-transitions and no color support. The first task was to modify the application to support these additional types. In PIPE there is no real separation between view and model. The view is responsible for containing all

model-information. A clear separation of view and model would have made the program structure a lot clearer.

After finishing work on the general Petri net presentation the next step was adding an editor for editing the currently selected nodes properties. The results of these efforts are shown in the following image.

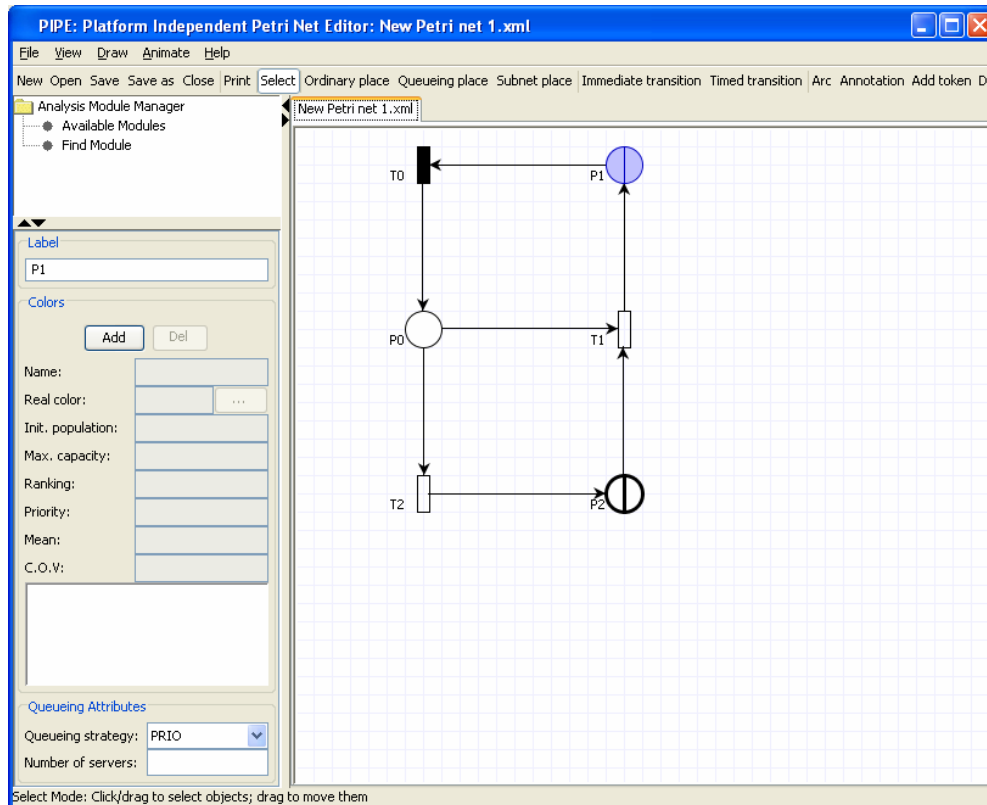


Image 10 PIPE-based QPE prototype

After finishing this extension the last remaining part was the incidence function editor. When starting to begin work it was clear that more invasive refactoring was needed. While performing this refactoring and hereby working on the PIPE base code it became obvious that even if the top level interfaces were totally object-oriented, the base was mainly procedural: Main classes consisting of numerous public static methods calling each other. Adding the needed functionality to support incidence-functions would have made refactoring the entire PIPE base necessary, which would have been more time-consuming than implementing the entire application from scratch, if to be done properly. The other attempt would have been to patch the existing implementation to support the additional functionality, but it would have been impossible for us to do, because it would have been a great conflict towards our understanding of software quality. This is why it was decided to stop work on the PIPE extension and implement everything from scratch using a visualization framework as visualization engine.

4.3 Frameworks

After deciding not to use PIPE it was decided to try using a visualization framework instead. These have been evolving to a quite mature state in the last few years. Since having come in contact with an application used for presenting mind-maps as net-structures in the course of our software-engineering lab course, we decided to have a closer look at the framework they used. This application was based on a library called Touchgraph.

4.3.1 Touchgraph based Approach

Touchgraph is a library used for creating editors to display complex network structures such as mind-maps, relations between people or information-fragments. The visualization and look and feel of Touchgraph applications is amazing. We were therefore eager to use this for creating the Petri net editor. The next image shows a snapshot of a Touchgraph based application.

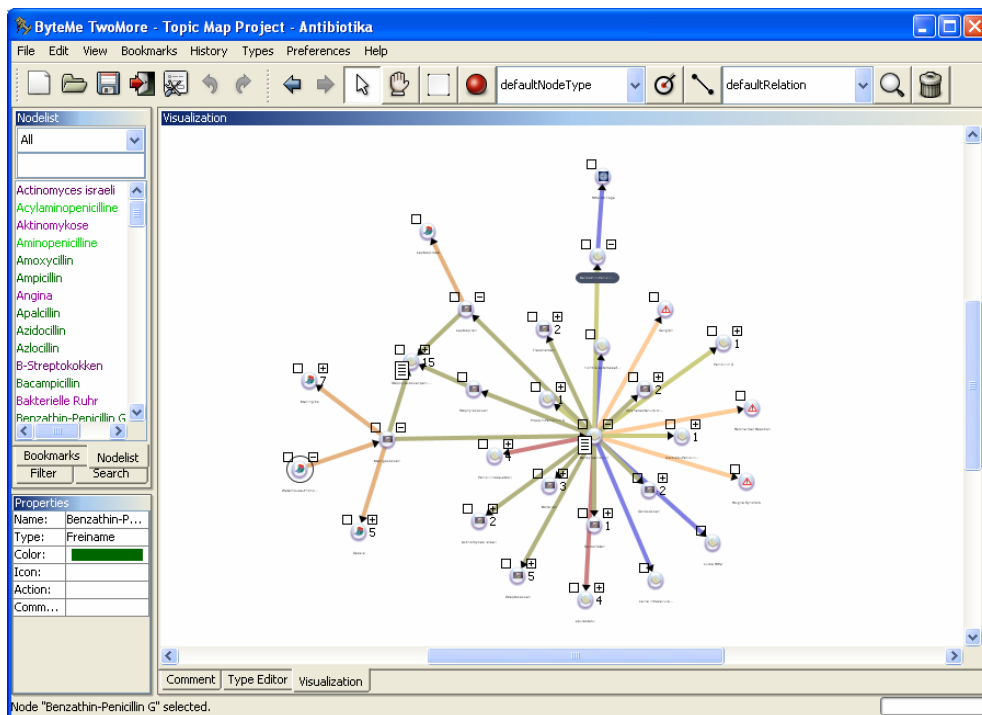


Image 11 Touchgraph application (TwoMore)

Unfortunately Touchgraph – as being a tool for editing networks of homogeneous elements – didn't support connection constraints such as:

- Preventing connections between places
- Preventing connections between transitions
- Preventing multiple connections between a source and target

Even if the elements could have different optical representations, it would have been extremely difficult to prevent connections between places or transitions or to add different attribute-sets to different types of nodes. The only reason for us not having continued on this path was the latest update of the Eclipse GEF sub-project which was an extremely promising approach.

4.3.2 Eclipse: RCP, SWT and GEF based Approach

Eclipse can be thought of as a platform for building software tools. It is not just an IDE for developing Java programs. Of course it is used for that, but it offers much more. It is a framework for integrating multiple tools within one application. While in earlier days the user needed a UML modeling tool, an IDE for the actual development, an XML editor and so on, each one with its individual project settings and look and feels, Eclipse now offers tool developers the possibility of integrating their services all within one platform. Now for example the Together UML editor is available as Eclipse plug-in, combining the power of TogetherSoft's UML editor with the Java IDE power of Eclipse. Altova's XMLSpy and other XML tools also integrate perfectly and to build up one universal platform for software development. Developers can now contribute own extensions to Eclipse.

With the Rich Content Platform (RCP) Eclipse offers the base Eclipse platform combined with a minimal set of plug-ins needed to build own stand-alone products based on the Eclipse framework. This eliminates the need to reinvent the wheel over and over again.

One major benefit of this approach is being able to use SWT. SWT is a technology competing against Java's Abstract Widget Toolkit (AWT) and SWING. AWT is restricted to offering only those controls supported on all platforms supporting Java. Those controls are directly linked to the operating-system's controls and therefore have an extremely high-performance. The downside is however being restricted to quite a small set of controls making complex GUIs with AWT almost impossible.

This lack of controls was eliminated by SWING by simulating all controls using primitive draw operations. With SWING it is possible to build relatively complex GUIs on any system supporting a Java virtual machine. Unfortunately this simulation of controls has its price in a relatively low performance. These SWING applications are mainly responsible for the general opinion that Java GUI-applications have a relatively sluggish performance.

Because of the lack of complex controls using AWT on the one hand and the bad performance of SWING, the Eclipse group had to go a different way:

If in SWT a control is supported by the underlying operating system this implementation is used directly. If not, it is simulated. It is clear that this library is restricted to the system it was developed for. The Windows library cannot be used on a UNIX system. This is why multiple versions were developed for each operating system that should be supported. Apart from the performance benefits of this approach there were several additional goodies. Now Java applications can gain access to parts of the operating system, they were never allowed to earlier.

For example it is now possible to place tray icons in the system tray. Participating in Drag & Drop actions is no longer a problem. Even embedding other Applications in a Java application using Windows OLE mechanism is possible now.

Generally it sounds like Eclipse, RCP and SWT should be all an application-developer always wanted. In general you could say “yes”. Unfortunately, Eclipse is extremely big and complex. When starting to develop Eclipse based applications the need of guidance is extremely high. Newbie’s can start relatively quick using tons of examples spread throughout the net. The major problems we encountered were big changes in the plug-in mechanism and the way things were done from Eclipse 3.0 to Eclipse 3.1. While starting work on the QPE application, Eclipse 3.1 was freshly released and therefore Google hadn’t indexed a lot of 3.1 content. Existing tutorials had to be partially rewritten, which hadn’t been done up to several months after releasing version 3.1. We sometimes had to spend days with debugging through Eclipse code to understand why and how things were done differently now.

Another problem with using examples to show a user how things are done, is that most of them don’t explain why something is done the way it is done, so the developer is limited to cloning a solution without understanding how it works. Most of the time the solutions provided by others only took me half the way to solving my problem, so multiple solutions had to be combined. Unfortunately, not all solutions are combinable, since there are different ways of doing almost everything in Eclipse and some techniques are incompatible with others. The result is that developers integrate code they don’t completely understand inside their application, which can be a really big problem when running into problems.

Another quite annoying side effect of the extremely high extendibility of Eclipse was that it was sometimes impossible to find out how components work from simply examining the code-samples. Most of the time, a deep look inside the Eclipse source code was required.

Take the `getAdapter` method for example, which very many objects in eclipse posses. It is used to give a component the chance of returning an adapter for different purposes. The purpose is defined by a parameter of type `Class`. If an outline-view is present for example, the `getAdapter` method is used to get an adapter for controlling this. If a property-sheet shall be displayed or the component should be able to provide an alternative property-sheet, this is done using the `getAdapter` method. Here again the only way to find out what adapters Eclipse is looking for, is setting breakpoints and have a direct look at the requests coming in.

The behavior of multiple components is controlled by implementing empty interfaces. Or sometimes implementing an interface opens an entirely different functionality. For really knowing what can be done with an object the user needs to debug Eclipses code and have a look at the interfaces being checked for. We used to joke with the comment that if Eclipse cost one cent per type check in its code it would be more expensive than any other development tool on the market. This is one thing that really started annoying us while developing the QPE application, since the solution would have been so simple, but it took hours and sometimes days for finding the right interface to implement. To make everything worse, sometimes multiple interfaces offer the same functionality on the first look, but result in entirely different behavior when being used.

To sum it up, Eclipse is missing some kind of big map helping the user understand why something has to be done the way it has to be. A clear definition and description of the possibilities and, what is extremely important: what the differences between the different approaches are.

It might seem that we would recommend avoiding using Eclipse after all the trouble we ran into, but we think the problems we addressed have partially been solved. Documentation quality is increasing rapidly and after getting to know the Eclipse platform we would definitely use it again, since the power it offers outweighs the cost by far.

5 QPE Tool Design & Architecture

A RCP (Rich Content Platform) is nothing else than an ordinary Eclipse installation without including any plug-ins. The developer can choose which plug-ins he needs and can manually add them to the plug-in dependencies. For example when creating an editor for editing text it might be useful to include the text-editor plug-in without any contributions and extend this with the desired additional functionality. Since the QPE application will contain a graphical editor which is based on the GEF (Graphical Editing Framework), this is one very important plug-in. For being able to use the intro-system for displaying the “getting started” content the `org.eclipse.ui.intro` plug-in is needed. Going into the details of every used plug-in would exceed the scope of this document, so we will simply summarize which plug-ins are used in the QPE application:

- `org.eclipse.core.runtime`
- `org.eclipse.core.resources`
- `org.eclipse.swt`
- `org.eclipse.gef`
- `org.eclipse.ui`
- `org.eclipse.ui.intro`
- `org.eclipse.ui.forms`
- `org.eclipse.ui.views`
- `org.eclipse.ui.editors`
- `org.eclipse.ui.console`

Generally the application consists of 2 GEF-based editors a set of actions, which contain the code that is executed whenever a menu or toolbar entry is selected and a plug-in integrating the SimQPN simulator into the application.

Since the package structure is built up according to the functional roles the classes have, the rest of the chapter is based on the systems package structure.

5.1 Plugin.xml

Even if developing a stand alone application. The `plugin.xml` file is the place where the application, perspectives, used plug-ins, menus, actions and much more is configured. It built up of a list of extension elements referencing so-called extension points. Explaining them all here and now would make no sense, so we will explain them where they are needed. Just keep in mind that there is this central configuration file.

5.2 Main application (de.tud.cs.qpe)

The root of the QPE packet structure contains two classes: Application and QPEBasePlugin

QPEBasePlugin is used as plug-in class which is used by the Eclipse runtime to get application icons and preferences as well as to define globally used constants. In case of the QPE editor it is additionally used to load image resources from the plug-in base directory using the getImageDescriptor method.

The Application class is the main class, which is executed when starting the application. All it does is basically call PlatformUI.createAndRunWorkbench and provide a WorkbenchAdvisor object to let RCP startup the application. This object is called ApplicationWorkbenchAdvisor in the QPE application. But how does Eclipse know that this is our main class? This is simply done by the org.eclipse.core.runtime.applications extension point element in the applications plugin.xml

```
<extension id="qpe.application"
    point="org.eclipse.core.runtime.applications">
    <application>
        <run class="de.tud.cs.qpe.Application"/>
    </application>
</extension>
```

5.3 Rich Content Platform (de.tud.cs.qpe.rcp)

As mentioned before when instantiating a new workbench, an ApplicationWorkbenchAdvisor object is provided. It provides multiple methods which are automatically called according to the applications lifecycle state. In the current case postStartup, restoreState and saveState are used. The methods saveState and restoreState are used to remember which documents were opened the last time the application was opened. And postStartup performs the task of actually opening the editors after the application is initialized. After opening the editors, postStartup initializes the RuleEngine which monitors open documents and checks them for errors and switches Stdout to write to the console view.

Apart from the lifecycle methods there is one further important one: getInitialWindowPerspectiveId. This returns the id of a perspective which should be used when starting the application for the first time. Which class initializes the perspective is defined in the plugin.xml file. In our case it tells the workbench to use the class de.tud.cs.qpe.rcp.Perspective.

```
<extension point="org.eclipse.ui.perspectives">
    <perspective name="de.tud.cs.qpe.perspective"
        class="de.tud.cs.qpe.rcp.Perspective"
        id="de.tud.cs.qpe.perspective"/>
</extension>
```

One very important method the WorkbenchAdvisor has to provide is the createWorkbenchWindowAdvisor method. This takes care of initializing the application's menus, toolbars and actions. But let's discuss this after dealing with the perspective.

5.3.1 Perspective

An Eclipse based application consists of workbench, perspectives, views and editors, where the workbench is the application itself. A perspective is an arrangement of visual tools, views usually provide additional information to the active editor and the editors contain the actual editing area. In our case, there is only one perspective. In other applications, perspectives are used to arrange the visual elements of the application according to different tasks the user wants to perform. In the Eclipse IDE for example there is a Java and a debug perspective, each with its individual layout presenting only the views needed. The next image shows a screenshot of an Eclipse version prior to 3.1. As soon as more than one perspective is active, controls for switching perspectives are shown. These are located at the top right side directly underneath the menu.

A typical Eclipse workbench is shown in the next image.

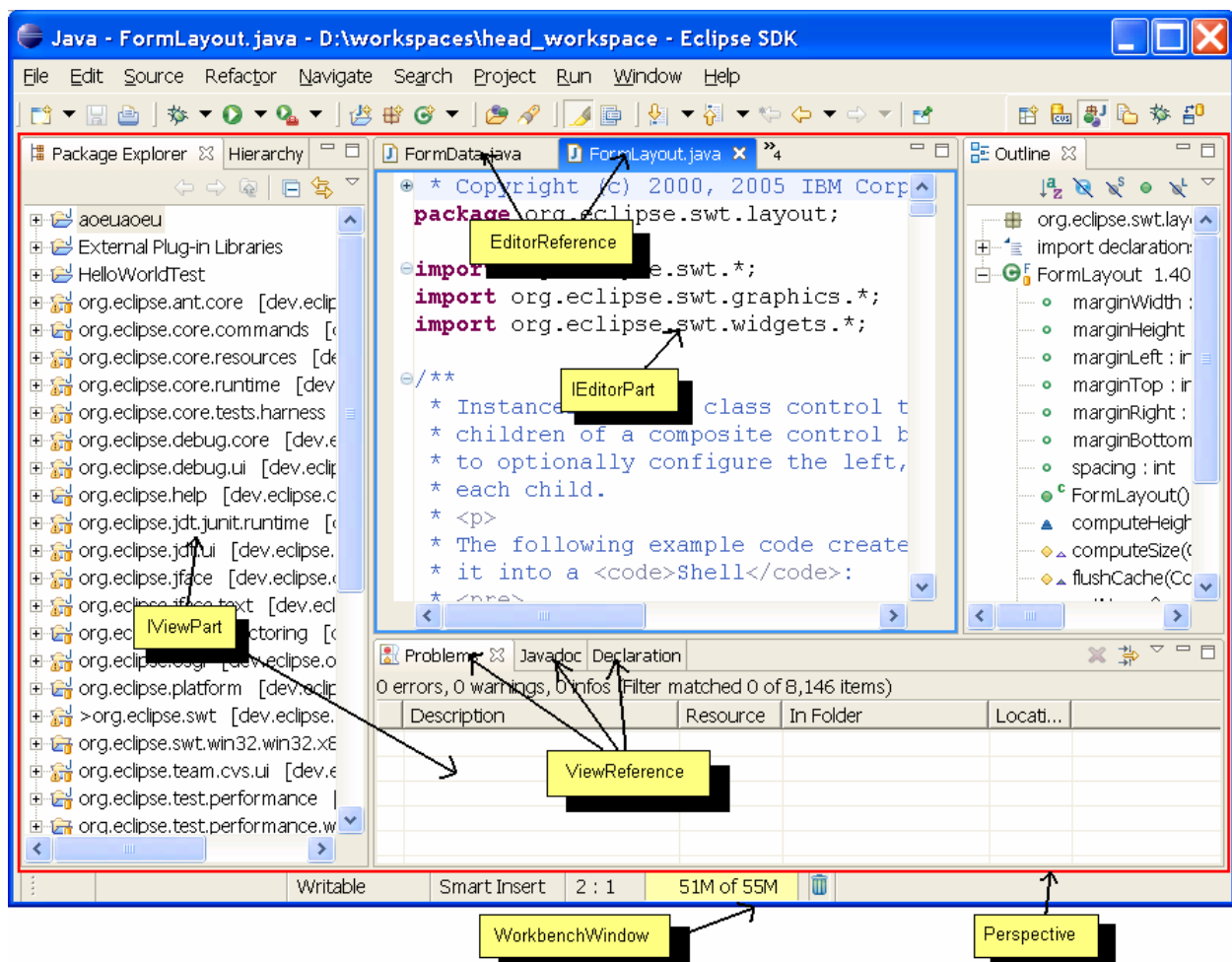


Image 12 Graphical elements of an Eclipse application

The Perspective class implements the IPerspectiveFactory and only contains a single method called createInitialLayout which programmatically shows positions and configures the views. In our case these are:

- Outline view: This offers the user direct access to the elements of the active editor. It is currently only used in the net editor.
- Property view: The property view shows the user a page of controls used to manipulate attributes of the currently selected element.
- Console view: Is used for reporting information to the user.
- Problem view: Is used by the RuleEngine to report problems or errors in the net. These problems could for example be places without colors, transitions without modes. Elements without incoming and outgoing connections. Initially using the standard Eclipse problem view was intended to be used for this. But problems using this in a RCP application made it necessary to develop a completely new View. This is done after finishing work on this thesis.

5.3.2 WorkbenchWindowAdvisor

The WorkbenchWindowAdvisor is responsible for initializing the application's window. For this it possesses similar methods as the WorkbenchAdvisor but these are linked to the application window's lifecycle instead of to that of the entire application. In the QPE application only the preWindowOpen lifecycle method is implemented. It controls the initial size of the application window, if a toolbar and a status line is to be shown.

In addition to the lifecycle methods there is one important method: createActionBarAdvisor. This creates an object that is directly responsible for initializing actions, menus and toolbars.

While normal Eclipse plug-ins usually hook in to an existing menu and tool bar structure in an RCP application the developer takes complete control over this aspect. This is done by the ActionBarAdvisor, called ApplicationActionBarAdvisor in the QPE application. It takes care of creating actions and building the menus, toolbars and so on.

During the application startup at first the ActionBarAdvisor's makeActions method is called to initialize the actions and then fillMenuBar and fillCoolBar are called to initialize the menus and tool bars.

The Actions initialized here are ones that operate without editors and are always available. Later on when discussing the editors, we will address this in more detail.

5.3.3 Actions (de.tud.cs.qpe.rcp.actions)

Actions represent objects being called as a result of a menu, toolbar, mouse click, context menu or other user-action. There is quite a large set of actions available directly from Eclipse. These are actions such as Zoom-In, Zoom-Out and an action for opening the intro screen. Most of them will be used in the editors and not in the global application since this only provides

Actions for creating, opening, closing and saving documents as well as the actions for showing and hiding the individual views present in the perspective.

Since actions can be used in multiple places they are not directly bound to a menu but are stored in an action registry. Wherever an action should be used its instance is retrieved by an id.

Creating and binding the action instances to the ids is what is done in the `ActionBarAdvisors.makeActions` method.

5.3.4 Menus and Toolbars

The application menus are built up in the `fillMenuBar` method of the `ActionBarAdvisor`. When called it is provided with an `IMenuManager` object, which can be thought of as the menu-root. To this object different menu managers or menu entries can be added. While adding a menu manager to an existing menu creates a submenu, adding an action adds a normal menu entry.

Since we are building the main menu at first a set of main menus has to be created. This is done by adding a menu manager for each menu. While creating a menu both a name and an id have to be provided. While the name is simply used as the menu's text, the id is used for referencing the menu. This is important if a user wants to additionally add entries to the menu using the `plugin.xml` extensions or one of the several extension mechanisms. If the user wants to divide the menu into different groups for better usability this is done using separators. When creating and adding a separator only a name has to be specified. This will not be displayed in the menu or toolbar, since it is used to reference the `Position` in the menu when adding additional item later on.

The `fillCoolBar` method initializes the part of the Application usually called a tool bar. The Eclipse guys seem to think their tool bars are cooler than all others, so they are called CoolBars. I'll just leave that statement uncommented.

5.4 Model (`de.tud.cs.qpe.model`)

In Eclipse all interaction between platform, view and model are handled by so-called `EditParts`. These are implemented by the developer himself. The structure and type of the model is therefore not subject to any restriction or convention. It's a common practice though, to use POJOs equipped with property-change-support, but that is not necessarily required.

When looking back at the entire development process, the model part might be the one part that was refactored most. In the initial version the model consisted of a set of POJOs (Plain old Java objects) extended with `PropertyChangeSupport`. This is the type of model used in almost all examples. While using this approach, we ran into serious problems when adding the Incidence Function Editor, since it used the same objects (e.g. Place objects) but in completely different context. A first solution was to split up the two editors into separate projects with the Incidence Function Editor being a plug-in for the net editor and both operating with completely alternate models. This added the requirement of code for casting and synchronizing the two models which was quite a nest for problems and errors.

After several attempts to sort out the problems we decided to do some radical refactoring.

Since the storage format would be XML, using an XML Dom Implementation as model sounded reasonable, except for the lack of any Dom implementation supporting event-based processing. One of the major benefits of using an Xml Dom as model is that by using XPath for navigating within the model the consistency checks and editor logic loses a lot of its complexity, since with POJOs the navigation and evaluations would have to be done manually. The only way to add event support to a Dom implementation would be to add property-change support to the chosen Dom implementation. Actually modifying an existing implementation would corrupt any upgrade ability, so we decided to simply wrap write-access to the Dom.

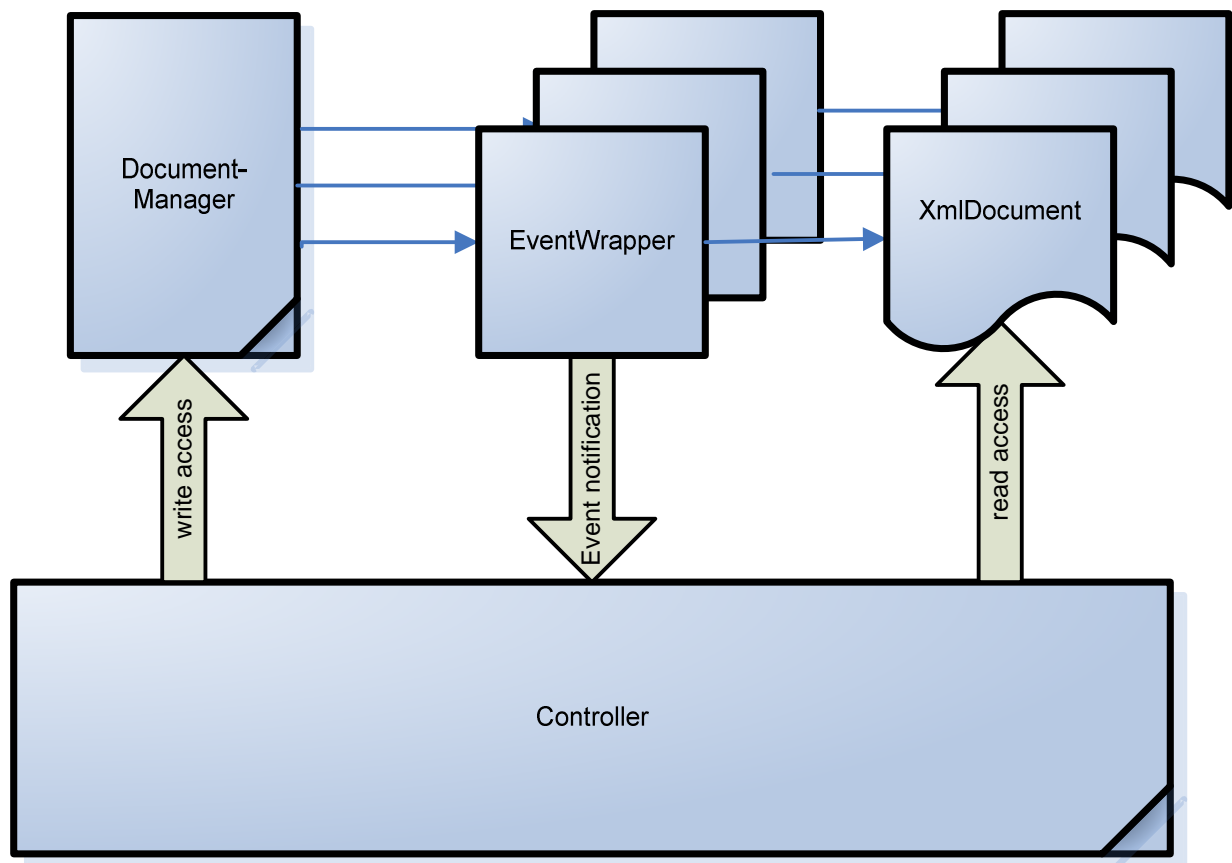


Image 13 Access paths using DocumentManager

This wrapping of write operation is handled by a component called DocumentManager. This provides a set of static methods for modifying documents. The paths used for accessing the model are shown by the previous image.

The process of registering a new document is illustrated by the following image.

When creating or loading a file, it is registered at the DocumentManager. The DocumentManager does this by generating an id (2) and stores this as event-manager-id attribute in the document being registered as well as an attribute signaling the dirty-state of the document. In a next step, an EventWrapper object is created and the document is assigned to this EventWrapper. The EventWrapper is then stored in the DocumentManagers document map with the event-manager-id used as key (3).

The EventWrapper is responsible for maintaining the elements event-subscription lists and for actually performing write operations to the document as well as firing events every time the document is modified.

After registering a new document the DocumentManger fires a DOCUMENT_REGISTERED event any listeners (4). In the current QPE application the only component interested in this type of event is the RuleEngine running consistency-checks on all opened documents.

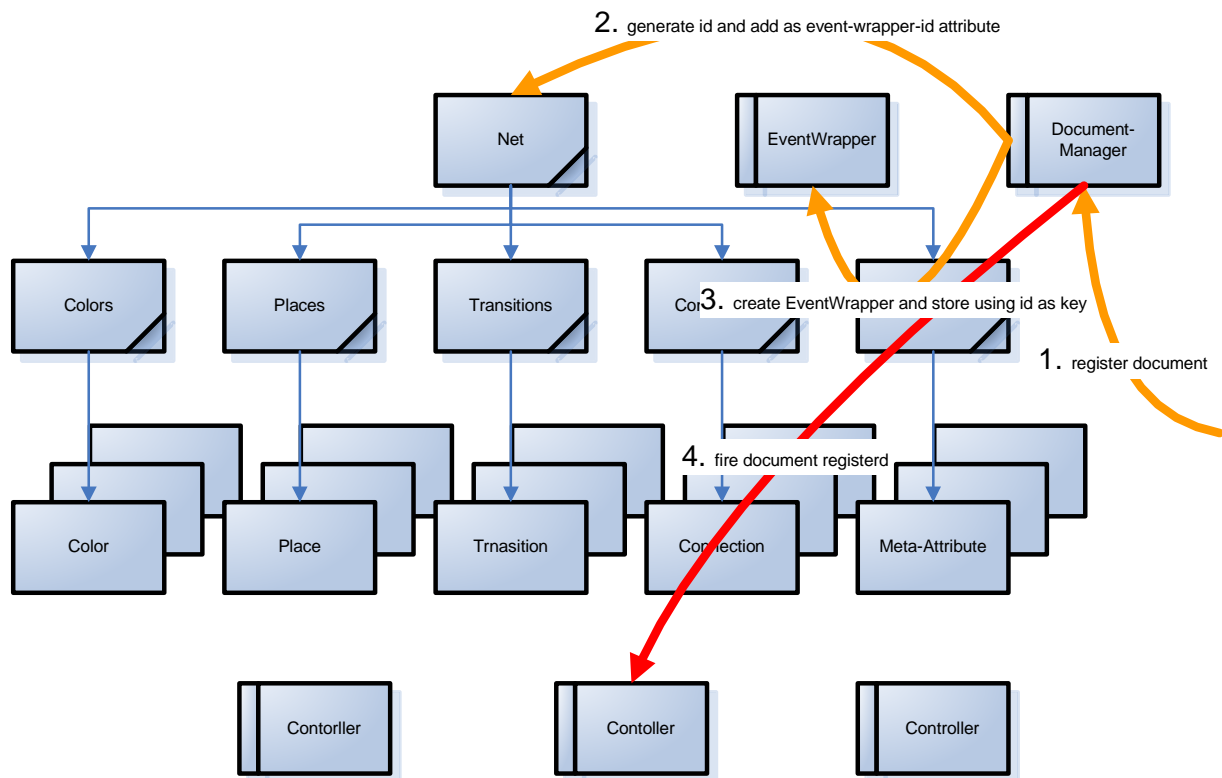


Image 14 Registering a new document

Any object implementing the `PropertyChangeListener` interface can now subscribe to document changes by calling the static `DocumentManager.addPropertyChangeListener` method (1). When calling this method, the element the client wants to listen for and a reference to itself has to be provided. The entire process of subscribing to elements change events is depicted in the next illustration.

In a next step the `DocumentManager` gets the event-manager-id of the document the element belongs to by calling the `getDocument` (3) method of the element and then using `getDocumentRoot` and `attributeValue` for accessing the event-manager-id (4). This is then used to lookup the `EventWrapper` object for that particular document in the `DocumentManagers` `EventWrapper-map`. After that the `addPropertyChangeListener` method is called on this object, which then adds the new listener to a list of listeners for that particular element (5). This list is then stored in a `HashMap` indexed by the element itself. This way by using an element as key the list of registered listeners can be retrieved.

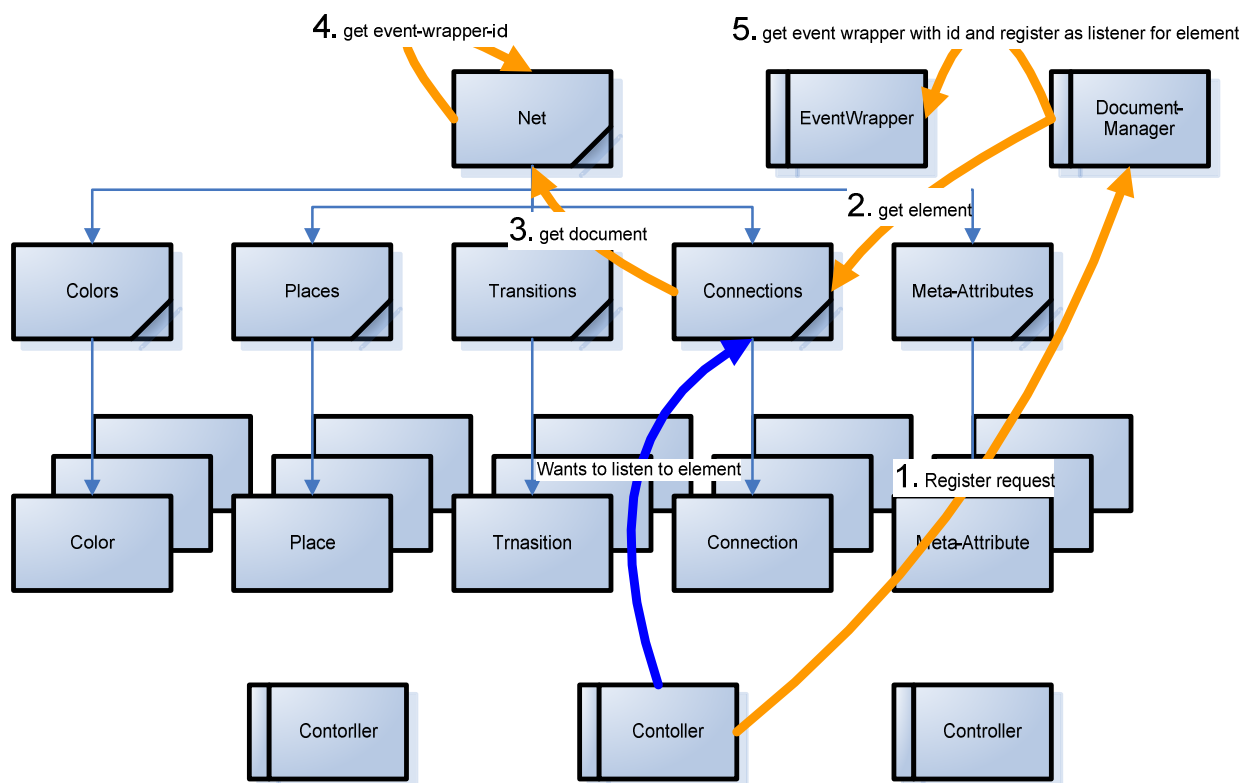


Image 15 Registering as change listener

When changing any node in the model the `DocumentManager` methods have to be used or the changes will be invisible to the listeners.

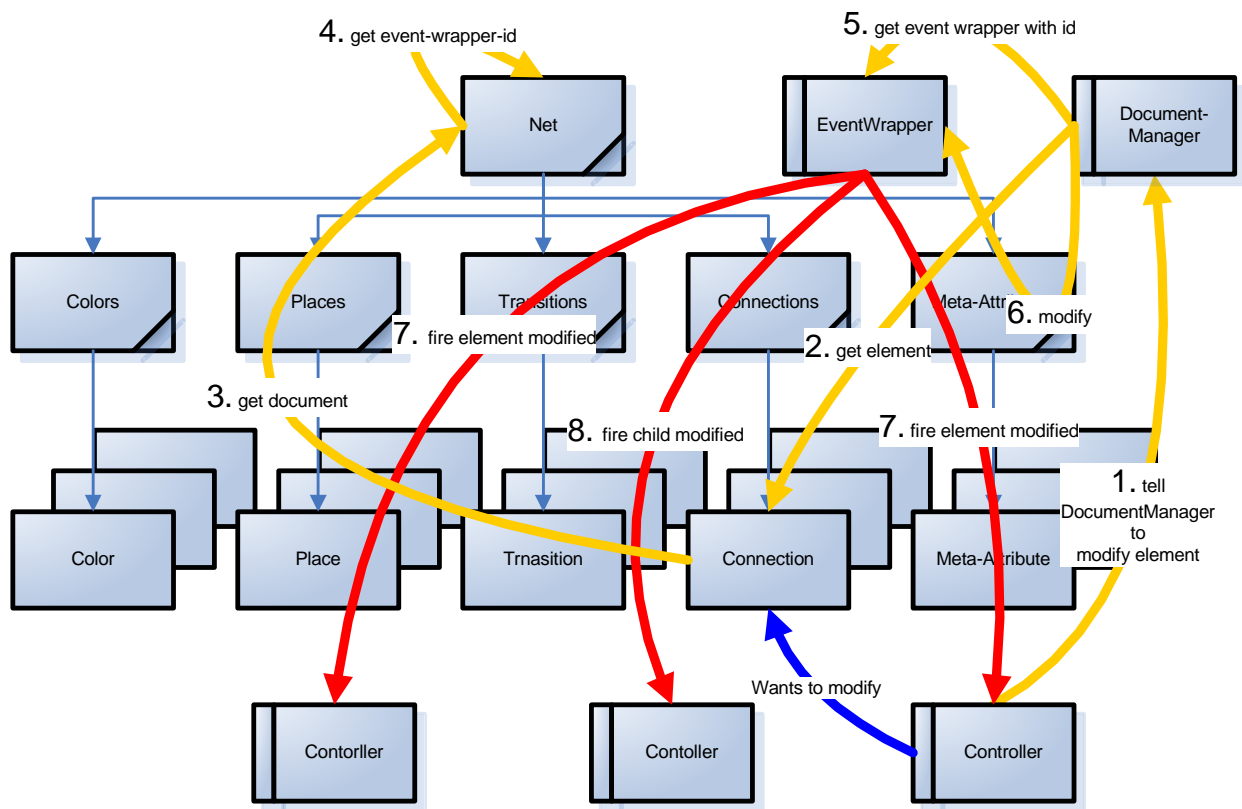


Image 16 Modifying a document

The above image shows the process of modifying an element using the DocumentManager.

If an object wants to modify an element then it has to do this using the corresponding method of the DocumentManager (1). The Wrapper is looked up the same way it is done for the `addPropertyChangeListener` method (2-5) and modifies the element using the corresponding event wrapper method (6). What happens now is that the event wrapper gets a list of event-listeners registered for this element using the registration `HashMap` and then fires a `CHILD_ADDED` event to any listeners registered for it (7). After that comes the really cool part, something that the classical approach using POJOs didn't support. After firing the `CHILD_ADDED` the event wrapper goes up the Dom tree firing a `CHILD_MODIFIED` for every parent up to the root (8). There the dirty-state attribute is set to dirty automatically. This allows us to listen to entire parts of the model without having to register to every individual element. In the net editor for example the only Element the Editor has to listen to is the nets root-element. This surely cleans up code dramatically.

As a final remark to the model we would like to mention that using the XML Dom approach cleaned up the model package from containing about 60 individual classes to one single public class and one private utility class (The wrapper).

The Dom implementation used was Dom4J because of its powerful API and good performance when dealing with large documents.

Since the model used in both editors are completely independent from the editor it is being used in, the DocumentManager was moved outside the package structures of the editors to one of the base packages instead.

5.5 Editors (de.tud.cs.qpe.editors)

In this subchapter the GEF based editors are described and explained. Because of the great similarity of the incidence and the net editor at first the parts are explained, that concern both editors after which the individual parts will be described.

At first it is important to explain how editors are defined and how they are activated.

```
<extension point="org.eclipse.ui.editors">
    <editor name="Net Editor"
        id="de.tud.cs.qpe.editor.net"
        icon="icons/sample.gif"
        class="de.tud.cs.qpe.editors.net.MultipageNetEditor"
        contributorClass="....net.gef.NetEditorActionBarContributor"
        default="false"/>
    <editor name="Incidence Function Editor"
        id="de.tud.cs.qpe.editor.incidence"
        icon="icons/sample.gif"
        class="de.tud.cs.qpe.editors.incidence.IncidenceFunctionEditor"
        contributorClass="..IncidenceFunctionEditorActionBarContributor"
        default="false"/>
</extension>
```

This segment of the plugin.xml configures the net and the incidence function editors. The most interesting attributes are the id, the class implementing the editor and contributorClass, which defines the ActionBarContributor used for this editor. The id is particularly important because it is used for creating the editors.

In the QPE application the net editor is opened in the NewAction and OpenAction classes:

```
IEditorInput input = new NetEditorInput(null);
try {
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().
        getActivePage().openEditor(input, NetEditorPage.ID, true);
} catch (PartInitException e) {
    e.printStackTrace();
}
```

The above code of the NewAction creates a new NetEditorInput object and then tells the Platform to open an editor with the given id. The id defined in the public static field of the editor is the same as defined in the plugin.xml.

The NetEditorInput class is nothing really special it simply takes a file name and tries to build a Dom4J Document from that files content. If the path parameter is null, a new document is created with freshly initialized structure (net, colors, places, transitions, connections and meta-attributes containers).

5.5.1 Gef (de.tud.cs.qpe.editors.???.gef)

A GEF editor is a special form of Eclipse editor. It is the linking-point of the Graphical Editing Framework and the Eclipse platform. Unfortunately, it also seems to constitute a breach in naming conventions and best practices. Inconsistencies that took quite some time to get used to. One that especially confused us was the definition of commands. While in Eclipse, a command is a logical structure without any implementation, actions are the places where the implementation is done. In GEF, commands are the only places which contain the logic. This caused quite some headaches up to the point of actually recognizing that the two Command base classes come from different packages and serve different purposes.

The net and the incidence function editors are implemented by classes extending the GraphicalEditorWithFlyoutPalette class. This creates a GEF editor which contains a so-called palette docked to the right-hand-side of the editor. In its initial state a small bar is displayed. As soon as the user goes over this bar, the palette opens and the tools inside are available for selection. Tools are arranged in so-called drawers which can independently be opened and closed. Initially all drawers are open. The net editor has a slightly different implementation structure, but this is described in the net editor chapter.

When opening a new editor very similar actions are performed as when starting the main application. For every editor there is a section in the plugin.xml where the editor is defined. In addition to the editors implementation a contributor class is defined, which enables the editor to contribute to the applications main menu and tool bar. As being extended from the same base-class its methods and their usage equal those of the main application window. One thing, which might be noteworthy, is that all actions being used here are actions implemented by the GEF framework and not custom implementations. But that doesn't mean that it isn't possible to use custom actions. They were just not used.

Even if we are going to cover commands and policies later on in this chapter it helps understanding the whole concept if we just imagined that commands contain the code for performing a certain task and objects called policies that generate command objects. These policies are registered using standard names.

These standard GEF actions now do nothing else then to access policy objects using their standard names and use one of the fitting policies methods to create a command object for performing the individual task. After that the command object is put into the system's command stack where it will be executed. This is why we can use a default copy Action for example, even if the implementation of the action has to be done individually for each editor. The places where the developer gains control of what is going to be done are the editors policies and of course the commands, but more to that later on in the chapter.

While the `ActionBarContributor` class is found and initialized by Eclipse's plug-in mechanism, the context menus are dealt with by the editor class itself. This sets a new instance of a custom `ContextMenuProvider` as provider for the current viewer in its `configureGraphicalViewer` method.

The `ContextMenuProvider` object has nothing more to do than to take the action registry provided to the object by its constructor and to hook-up the actions in the context menu using the `buildContextMenu` method.

GEF is based entirely on the Model View Controller (MVC) concept. There are no rules for developing the model but the rules for developing both controller and view components are quite complex. This is why, after explaining the initialization and purpose of palette and property view, the view and controller part of GEF are explained in detail.

5.5.1.1 The palette (`de.tud.cs.qpe.editors.???gef.palette`)

The code for initializing the palette basically consists of a factory for creating the palette entries and a set of template-classes being used for creating new editor objects.

The factory used for building the drawers and tool entries of the palette is built up to create everything when calling the `createPalette` method. This class contains a protected method to initialize each drawer.

In case of the QPE application there are three different types of palette entries: selection tools, creation tools and connection tools. The only selection tool is the `PanningSelectionToolEntry` offering a tool to select single elements or a whole area of elements.

Most elements in the net editor's palette are `CombinedTemplateCreationEntries`. These take additional parameters in their constructor. Apart from label, images and tooltips they take a parameter of type "Class" which is used as template for new elements and a factory of type `CreationFactory` which will be used to create object instances for a given template class parameter.

The connection tool (`ConnectionCreationToolEntry`) is responsible for creating connections between two elements. Since creating connections demands a little different handling, this is why there is an extra connection creation tool.

Since this is the first time images have to be loaded and this is a little trickier than usual we will just briefly explain how loading the image-resources for the palettes is done. Here comes the code for this part:

```
public static ImageDescriptor getImageDescriptor(final String fileName) {
    Bundle bundle = Platform.getBundle(QPEBasePlugin.PLUGIN_ID);
    final URL installURL = Platform.find(bundle, new Path("/"));
    try {
        final URL url = new URL(installURL, fileName);
        return ImageDescriptor.createFromURL(url);
    } catch (MalformedURLException mue) {
        return null;
    }
}
```

This method is provided with a relative filename. As a first step a bundle name is looked up. A bundle can be thought of as an internal id Eclipse uses to reference an applications or plug-ins resources. Usually it consists of a number followed by the plug-in id. After retrieving the bundle name the platform can be used to get the installation directory of the plug-in. Now all that has to be done is to add the relative path to the absolute plug-in path and the resources can be normally read.

5.5.1.2 The property view (de.tud.cs.qpe.editors.???.gef.property)

The property view is a special view showing properties for the currently selected element in the editor. Per default it is a plain two-column table with very limited grouping functions. This is why we decided to implement my own property-view. The following images show both the standard an QPE version of the property view.

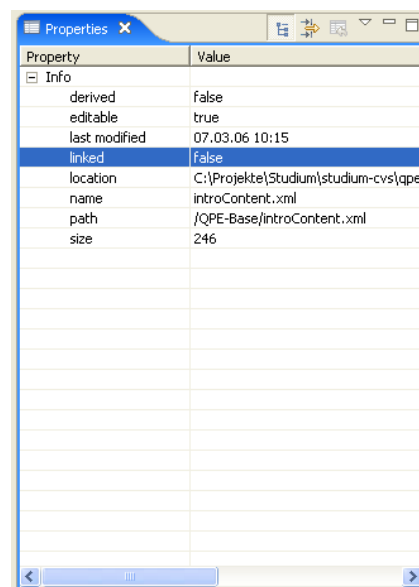


Image 17 Standard Eclipse property view

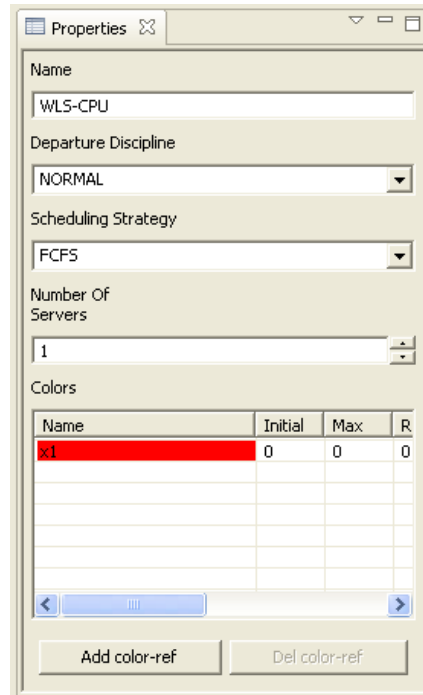


Image 18 Custom QPE property view

The tricky part is to tell Eclipse not to use the default one and use the QPE one instead. This is done with the `getAdapter` method of the `NetEditorPage` or `IncidenceFunctionEditor` class. As soon as a view is added to the perspective Eclipse starts asking for custom adapters for this view. This is done by calling the `getAdapter` method of a component and by passing a class parameter to it. During initialization Eclipse asks the new editor for a property view adapter by using the `IPropertySheetPage` class as parameter.

In case of the property sheet an object implementing the `IPropertySheetPage` interface is returned, which then replaces the default view. The `IPropertySheetPage` interface defines two important methods:

The `createControl` method is used for actually setting-up the property page. In our case, since it harbors the property pages for multiple object types it consists of a simple composite with a `StackLayout` `LayoutStrategy`. This adds all child elements behind each other and always shows only one. The children of this composite now are the actually shown property pages.

The second method is the `selectionChanged` method. As mentioned before, the property sheet shows properties of the currently selected elements. This method is called every time a selection changes. It is responsible for un-registering the property change listeners for the previously selected node, register for property changes of the new one and to finally pop the property page for the newly selected node type to the top of the stack.

The individual property pages are just simple SWT composite objects with controls for editing the corresponding node type's properties.

5.5.2 Controller (de.tud.cs.qpe.editors.???. controller)

All controller objects are called EditParts depending on their type they are derived from AbstractGraphicalEditPart for ordinary elements or AbstractConnectionEditPart for connections. It is their job to keep model and view in sync and communicate with the rest of Eclipse. All events and actions that apply to a visualization part or model go to the EditPart which is responsible for updating everything.

The EditParts are all created using an EditPartFactory. This is set in the editor's initialization phase in the editor's configureGraphicalViewer method. This factory has only one method that has to be implemented: getPartForElement. This method creates an EditPart instance suitable for managing the model object which was passed as a parameter. After that it assigns this model object to the newly created EditPart using that EditParts setModel method.

Edit parts have 3 important methods to implement:

- createFigure,
- createEditPolicies and
- getModelChildren

The first is for creating a graphical representation of the EditParts model. What the others are used for will be described in the rest of this chapter.

EditParts can have child EditParts, in our case the root EditPart, which is the global editors EditPart, has multiple children which represent the editor's content elements. In order to get these child EditParts an EditPart provides a getModelChildren method returning a list of child elements – a list of child model elements not EditParts. In order to get the EditParts, for every model element the EditPartFactory is used to generate the proper EditPart.

How the EditParts delegate responsibility is implementation dependant. In general the root EditPart could deal with synchronizing all model and view elements, but that would make it much more complex. The other extreme would be to let everything be handled by children. In this case we could have an extra EditPart for every Dom element in the model – even the containers such as “colors”, “places” and so on. But that would add lots of unnecessary code (in the wrappers EditParts, since having no graphical representation, they would have nothing to manage). In the following Diagram the EditPart structures used in the net and incidence function editor are shown.

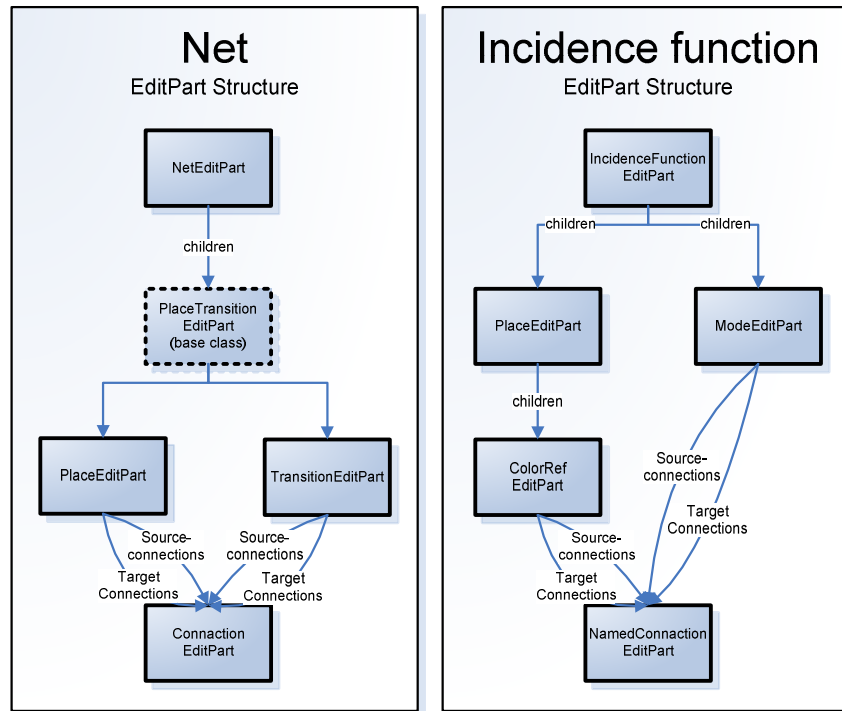


Image 19 EditPart structures

How the EditPart react on platform events is handled by EditPolicy objects. These are initialized in the createEditPolicies method. The EditPart can install multiple EditPolicies using the installEditPolicy method. Depending on an EditParts role different EditPolicies can be installed. This role-name has to be passed as additional parameter with the policy object itself to the installEditPolicy method.

For example in the net editor place and transitions are allowed to accept connections. In order to accept new connections a GraphicalNodeEditPolicy is bound to the role GRAPHICAL_NODE_ROLE. This Role handles connecting and reconnecting source and target connections to the current Element. In order to delete connections an additional policy has to be installed. For this a ConnectionEditPolicy is bound to the CONNECTION_ROLE. The reason for not dealing with all in one policy is that for deleting only knowledge of the current node. For all others additional information is needed.

For the root EditParts the EditPolicy RootComponentEditPolicy is installed. This is only responsible for preventing the current element from being removed from its parent. Since being the root it has no parent.

Connections bind a ConnectionEndpointEditPolicy to the CONNECTION_ENDPOINTS_ROLE for managing connecting and reconnecting the endpoints of the current connection and bind a ConnectionEditPolicy to CONNECTION_ROLE to deal with connection removal.

Each type of EditPolicies offers a different set of methods. These are called by the actions created by the ActionBarContributors to retrieve a command object containing the logic for actually performing the action.

Commands extend the base Command class. I'll simply explain the functionality of commands with the ConnectionCreateCommand which is used for creating connections.

When clicking in the editor for creating a new connection, the connection action calls the getConnectionCreateCommand method of that EditParts EditPolicy, creating a ConnectionCreateCommand with the source set to the source model. This command is returned and is used by the connection create action.

As soon as the mouse is over a new EditPart now that EditParts EditPolicy getConnectionCompleteCommand method is called. When calling this method a CreateConnectionRequest is passed to the method. This object is used to retrieve the original ConnectionCreateCommand. In this object the target property is set to the new EditParts model. After that the command is simply returned and no new command object is created.

Whenever something changes in the editor, even if it is just the mouse pointer position, the ConnectionCreateCommands canExecute method is called. It is this methods only purpose to check if all constraints are met. In case of the connection creation only connections from place to transition or transition to place are allowed. Additionally only one connection is allowed for each direction. If no constraint is violated the method returns true. This can be seen in the editor as the mouse pointer loses the no-entry sign. As soon as the user now clicks and canExecute returns true, the execute method is called to actually do the task of executing the command by persisting the changes in the model, in this case by adding a connection element to the net.

After the command was executed the command object is stored in the command-stack. This is used to undo the commands changes. For this the command can implement the undo method. If the command is not undoable additionally the canUndo method has to be implemented to return false. It is then not added to the undo stack.

Most other Commands are a lot simpler since they are no multi-step commands. When adding new elements to a net for example, the NetEditParts EditPolicies getCreateCommand method is called every time the mouse moves, initializing the PlaceTransitionCreateCommand with the new element and the bounds containing the current mouse position. Immediately after that the canExecute is called. As soon as the user clicks in the editor area the execute method adds the new element to the net. No further steps are needed.

This creation and destruction of objects for each pixel the mouse moves was one thing we think the Eclipse developers could have solved a little prettier.

Once an EditPart is created and a model object is assigned to it, the EditParts activate method is called, which makes the EditPart register itself as PropertyChangeListener to the model object. Whenever the model changes it is notified and it can update the view elements.

When an EditPart is destroyed because the element is deleted or an element containing it was destroyed (maybe the editor was closed) the deactivate method is called making the EditPart unregister from the model element.

5.5.3 View (de.tud.cs.qpe.editors.???. view)

The view implementations of the net editor and incidence function editor have little in common so this chapter will be rather short.

Figures have bounds. This can be thought of as a rectangular area where the figure can draw to. If a figure has child figures, these can only draw to the area defined by the parent. This is especially interesting in the incidence function editor.

Besides the bounds it has some handle-bounds. These are only used in the net editor. Whenever a user clicks on an element (or selects multiple) using the selection tool. It is highlighted like in the following picture:



Image 20 Selected net editor element

The boxes on the border usually are used to resize the figure, but since in the QPE editor resizing of places and transitions is not supported, they are for decoration purposes only.

Usually these handle-bounds are equal to the bounds of the figure. In the above sample this would not be what we want. Since the label belongs to the figure, the handle-bounds would have surrounded the text and the circle. In order to take control over the handle-bounds a figure can implement the `HandleBounds` interface which defines a single method: `getHandleBounds`. The developer can use this to return alternate bounds. In case of the above Place the bounds returned by the `getHandleBounds` are the bounds of the circular child figure.

The next special thing about figures is that, if they participate in connections, the system has to somehow determine where the tail and head of the connection arrows have to end. Simply letting them end in the middle would have been quite ugly. For taking control over this aspect, the so-called connection anchors are used. These calculate where the arrow ends depending on from where it comes from. In GEF there can be a difference between incoming and outgoing connections. For example, places could be created in a way, that outgoing connections start in the middle or at the top and incoming connections stop at the figures border. For the place, transition and color-ref figures it was not necessary to implement own connection anchors since ones for rectangles and ellipses already existed. One specialty is the mode figure, as for this a custom connection anchor had to be developed.

5.5.4 Net editor (de.tud.cs.qpe.editors.net)

5.5.4.1 MultipageEditor

There is an `IncidenceFunctionEditor` class but no `NetEditor` class in the package structure of the QPE application. This is because of the nature of the net editor. Since the definition of colors

was separated from the places there was need to add an additional editor: the color editor. Instead of creating a whole new editor, both editors needed for creating a net were bundled in one MultipageEditor this editor is called MultipageNetEditor.

A MultipageEditor is no real editor in the common sense. It is much more a structure able to contain multiple editors. These are called EditorPages. For each editor page a tab is added at the bottom of the editor area. The MultipageEditor contains two editors. It contains a NetEditorPage and a ColorEditorPage. First contains the GEF editor for editing the net and the second contains the editor for editing the color-definitions for the current net.

The editors content is setup using the createPages method which indirectly instantiates the NetEditPage and a ColorEditorPage and adds them to the MultipageEditors page-list using the addPage method.

Since the net editor is to be considered the main editor and all selection changes are sent to this, all getAdapter calls are also redirected to this. This can be seen when switching to the color editor that the property page is not updated and continues to show the properties of the previously selected element of the net editor.

The NetEditorPages getAdapter method reacts on requests for the property page and the outline view. This is not active when using the incidence function editor.

Even if this MultipageEditor is quite a nice thing, it adds some really nasty problems to the application – problems that took days for sorting out. One of the biggest problems was that in the incidence function editor my actions in the menus, toolbars and context menu nicely updated their activation state when changing the selection inside the editor. They didn't do that in the net editor. After quite a while of searching we found out that the MultipageEditor as being the parent of the NetEditorPage is identified as source of the selection change and therefore the NetEditorPage thinks the selection change came from somewhere else so no updating has to be done. To make the NetEditorPage accept the change events the MultipageNetEditor has to explicitly implement the ISelectionListener interface and repost the selection change with the NetEditorPage as source. Here comes the code fragment fixing the selection problems:

```
public void selectionChanged(IWorkbenchPart part, ISelection selection) {
    if (this.equals(getSite().getPage().getActiveEditor())) {
        if (netEditor.equals(getActiveEditor())) {
            netEditor.selectionChanged(getActiveEditor(), selection);
        }
    }
}
```

This catches all selection change events and if the selection originates from the active editor and the net editor page is the active page of the MultipageEditor, the selection changed is explicitly called on that with the source set to the editor itself. After adding this code activation and deactivation worked like a charm. This was one problem that would have taken days for solving if we hadn't received a tip in the Eclipse GEF newsgroup.

5.5.4.2 NetLayoutEditPolicy

In order to react to adding, removing and moving elements inside the net editor an additional EditPolicy is bound to the LAYOUT_ROLE. The XYLayoutEditPolicy offers methods for creating creation, deletion, movement and resize commands.

5.5.5 Incidence function editor (de.tud.cs.qpe.editors.incidence)

One of the probably biggest differences in the incidence function editor is that the user is not allowed to change the position of any element in the editing-area. When being resized it dynamically layouts itself and contains more complex figures than the net editor. The last two aspects caused the biggest problems.

5.5.5.1 Model (de.tud.cs.qpe.editors.incidence.model)

As mentioned before an EditParts getModelChildren method is used to get the children of an EditPart. Since a place can be input place and output place of a transition at the same time this causes quite some problems. Two references to the same Dom element would have to be returned. Unfortunately, the Java class List used as return value does not allow adding two references to the same object in one List object.

Another problem is how to distinguish between input places and output places, since there is nothing the editor could use as a hint. Referencing the same object the models for both are absolutely identical. Inside the getModelChildren method it is however no problem to distinguish between the two types, since their type is defined by the type of connection used for getting their reference.

The solution to both problems was a quite simple one: Simply by wrapping the element references using wrapper objects, the duplicate list entry problem is solved. By using two types of wrappers distinguishing between the two types of elements is no longer a problem.

The following image illustrates the wrapping of input and output places in the incidence function editor.

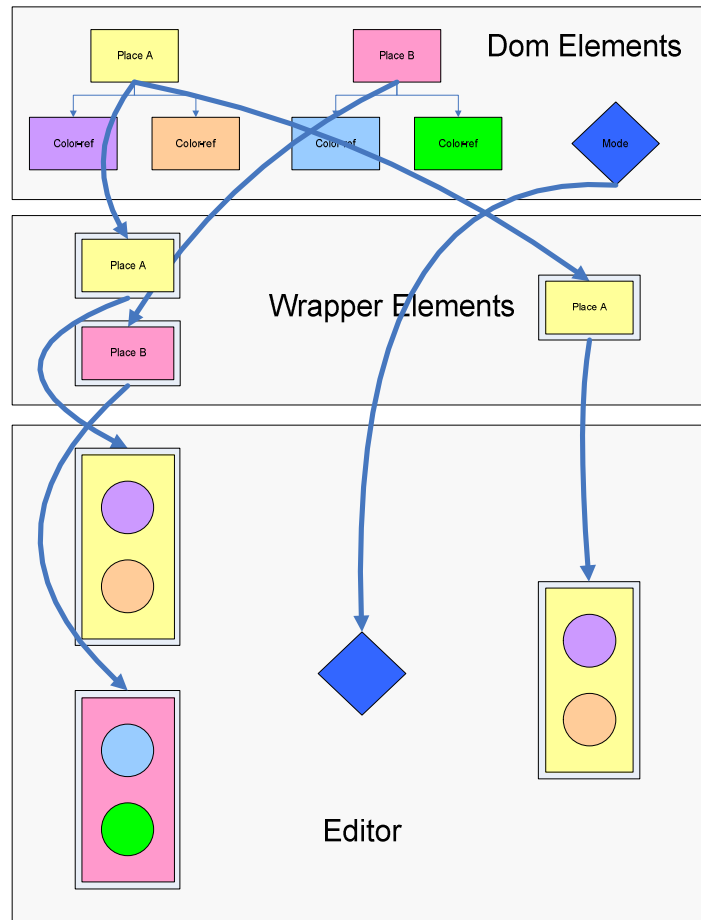


Image 21 Wrapped element references

5.5.5.2 Model (de.tud.cs.qpe.editors.incidence.view.layout)

First of all the layout has to be described and automated somehow. This is done by the `IncidenceFunctionLayout` class. This is set in the `IncidenceFunctionEditorParts` `getFigure` method. All it does is take all children of the editor, calculate some sizes and then to update the x and y positions of every element in the editor.

The layout process could be defined as the following: At first the elements are sorted by input place, mode and output place. Then all children are told to do their own layout. This is especially important for the input and output places since they have to arrange and resize in order to correctly display their color-ref child figures. After this is done the child elements all have their correct sizes and calculating the positions can be started. 100% height is assumed as the maximum height one of the three categories need. The other categories margins are updated in a way that they fill up the vertical space equally distributed.

The horizontal alignment is relatively simple to that. Input places are aligned to the left-hand side, output places on the left-hand and finally modes are placed exactly in the center of the editor area.

5.5.5.3 View (de.tud.cs.qpe.editors.incidence.view)

The view part is a little more complicated as with the net editor, as the incidence function contains figures that are composed of multiple child figures, as shown in the following image.



Image 22 Figure of place with 5 color-refs

The PlaceFigure which is created by a PlaceEditPart is based on the BaseFigure as almost all other figures in QPE. It contains a createFigure method used for creating the figure's Figure object which is equipped with a label underneath it. This PlaceFigure is basically the same as an immediate transition figure except that the fill color is a different one and a layout manager has been set to layout child elements underneath each other inside the place-rectangle's bounds. The child figures are added by the EditPart automatically.

The color-refs themselves are a lot like the figures of ordinary places except that their fill-colors can be set and that the labels are positioned right to the circular figure.

When creating the EditParts for the places children their figures are automatically added to the places figure and the layout manager of that makes the color-ref figures align underneath each other nicely.

5.5.5.4 View (de.tud.cs.qpe.editors.incidence.view.anchor)

As mentioned earlier in the text for places, transitions and color-refs default implementations of connection anchors could be used. In case of modes this was no longer possible, so a custom connection anchor had to be implemented.

The ModeAnchor extends AbstractConnectionAnchor and is used to calculate the position of the connections end. It contains three methods that should be of interest to us:

- getLocation,
- getBox and

- `getReferencePoint`

The method `getBox` simply returns the bounds of the figure this anchor belongs to. `getReferencePoint` returns the position of the center of the bounds returned by `getBox`. Finally, `getLocation` is the place where calculating the endpoint position is done. The result simply describes that point's location.

Implementing the `getLocation` was quite interesting since we had to dig out our old high school math books and develop the calculation on my own.

5.6 Rule Manager

The rule manager is a component which is intended to run validity checks against opened and changed documents and to report them in the problem view. Unfortunately, there are some major problems with the problem-view. These problems necessitate creating a custom problem view. But since implementing such a view will probably result in big problems and the time was running away, it was decided to switch to reporting errors to the console until this view is finished after finishing this thesis. We didn't want to skip mentioning the RuleEngine because of its benefits for the entire Application.

While editing a net or incidence function a lot of things can be done wrong, which can or should not be caught by application logic. For example when adding a new place this place initially has no color-refs defined. If a mode is added this too has no modes defined initially. If trying to edit an incidence function of a transition with no modes or a place with no color-refs, this makes no sense and should be reported by the rule manager.

The rule manger can be thought of as a service running inside the QPE application. It is instantiated in the `ApplicationWorkbenchAdvisors` `postStartup` method.

When created the rule manager registers itself as listener for `DOCUMENT_ADDED` events at the `DocumentManager`. These events are the only type being directly fired by the `DocumentManager`. When a document is loaded the rule manager adds itself as change listener to that document. Every time the document is modified it runs a set of tests defined in an external XML document containing the rule definition against the document that was changed. For every rule that failed an error is reported.

At the moment whenever a rule fails log output is written to the console view as proof of concept but is currently commented out since the SimQPN plug-in also logs to that view and when writing back test run results or changing the simulation settings the rule managers output would corrupt the output of the SimQPN plug-in.

5.7 Problems

While developing the application we ran into a number of problems, which took us a lot of time to solve. This chapter deals with the issues causing the most problems and how these problems were solved.

5.7.1 Unwanted menu and toolbar entries

In order to redirect the output of any tool to the console view, this plug-in had to be included to the application. Because of the console plug-in's dependency on the text editor plug-in, this had to be added too. Unfortunately, while initializing, this plug-in writes quite a lot of error messages and exceptions to the log. All of these errors occur because the text editor plug-in tries to add menu entries to menus that don't exist. The only way of getting rid of the errors was to add the menus the plug-in was looking for.

This had the side effect of adding some annoying and unusable menu entries to the application menu. Since reimplementing the console view without the text editor view is something the Eclipse developers are working on for quite a while, we solved the problem by simply adding the menus and hiding them. In order to still be able to display the actions in the menus, new menus with the same name but different ids had to be created. Even if this is quite an ugly hack, it is the only solution to having both the console view and no errors or phantom menu entries.

As soon as a console view that operates without the text editor plug-in is available we will switch to that and remove the hack.

5.7.2 Menu entries that can be checked

Eclipse allows an action to be checked. A checked action has a checked checkbox symbol left to its text in menus. For this the action class offers a method called `setChecked`. It turned out that even if Eclipse offers this functionality, it has no effect. Checked actions were never shown checked in the menus. This was needed for the entries of the view-menu. After quite a while of searching it turned out that this functionality is only available if the actions were not programmatically added using the `ActionBarAdvisors` but using the plug-in mechanism using the `plugin.xml` file.

```
<extension point="org.eclipse.ui.actionSets">
  <actionSet label="View Actions" visible="true"
    id="de.tud.cs.qpe.actionset.view">
    <action label="Outline"
      class="de.tud.cs.qpe.rcp.actions.window.OutlineViewDelegate"
      tooltip="Shows/Hides The Outlines View"
      menubarPath="qpeView/viewMenuGroup"
      id="de.tud.cs.qpe.view.actions.outlineAction"/>
    <action label="Properties"
      class="de.tud.cs.qpe.rcp.actions.window.PropertiesViewDelegate"
      tooltip="Shows/Hides The Properties View"
      menubarPath="qpeView/viewMenuGroup"
```

```

        id="de.tud.cs.qpe.view.actions.propertiesAction"/>
<action label="Tasks"
        class="de.tud.cs.qpe.rcp.actions.window.TaskViewDelegate"
        tooltip="Shows/Hides The Tasks View"
        menubarPath="qpeView/viewMenuGroup"
        id="de.tud.cs.qpe.view.actions.tasksAction"/>
<action label="Console"
        class="de.tud.cs.qpe.rcp.actions.window.ConsoleViewDelegate"
        tooltip="Shows/Hides The Console View"
        menubarPath="qpeView/viewMenuGroup"
        id="de.tud.cs.qpe.view.actions.consoleAction"/>
</actionSet>
</extension>

```

The label attribute represents the label the action will have in the menu. Class specifies the class implementing the action. Tooltip is shown if the mouse hovers over the menu entry for some time. The interesting part here is the menubarPath parameter. The first part specifies the id of the menu these actions will be added to. In this case they will be added to the menu with the id “qpeView”. This was defined in the main application ActionBarAdvisor. The second part specifies a separator group. If this group already exists the action will be added to it. If not it is created. In the above example all actions will be added to the same group. Finally the action received an id so it can be referenced in the system. This can for example be used to add the action to a tool-bar or context-menu.

This part of the plugin.xml does the same as programmatically adding the actions. The actions themselves however have to be implemented a little differently.

These actions do not extend Action anymore but have to extend the ActionDelegate class. These classes’ methods signatures are extended by passing an IAction object. It took quite some time understanding how the actions mechanism works. During application startup the actions defined in the plugin.xml create standard multipurpose action-objects without any implementation of the action logic. These action objects are configured using the parameters of the plugin.xml. As soon as they are activated the delegate class defined in the plugin.xml is instantiated and the dynamically generated action object is passed as parameter to the delegates methods. This way the delegate can access the IActions methods as if it would have accessed its own ones when being implemented as Action. The only difference is that calls to setChacked method work this time.

5.7.3 Adding elements to context menus

If actions should be added to editor’s context menus the only working way to do this is using the plugin.xml again. Even if it should be possible doing this programmatically, it never worked properly. Here the segments of the plugin.xml:

```

<extension point="org.eclipse.ui.popupMenus">
    <objectContribution id="de.tud.cs.qpe.editor.net.popup"
        objectClass="...net.controller.editpart.editor.TransitionEditPart">
        <action id="de.tud.cs.qpe.editor.net.popup.showProperties"
            class="...net.gef.action.OpenIncidenceFunctionDelegate"

```

```

        menubarPath="elementActions"
        label="Edit Incidence Function"/>
    </objectContribution>
    <objectContribution id="de.tud.cs.qpe.editor.net.popup"
    objectClass="...net.controller.editpart.editor.PlaceTransitionEditPart">
        <action id="de.tud.cs.qpe.editor.net.popup.showProperties"
        class="...net.gef.action.ShowPropertyDelegate"
        menubarPath="properties"
        label="Properties..." />
    </objectContribution>
</extension>

```

The `org.eclipse.ui.popupMenus` indicates that context menus will be defined here. The tag `objectContribution` tells Eclipse that this context menu element will be bound to objects inside the editor. In contrast to this `editorContributions` are available in the entire editor. The most interesting attribute here is the `objectClass` attribute. It tells Eclipse for which elements the action is available. In the above example the “Edit Incidence Function” action is available only over `TransitionEditPart` objects. Even if the object visible in the editor is actually the figure, the `EditPart` is as mentioned before the component responsible for communicating with the platform. `Label` defines the text used in the menu entry and `menubarPath` the place where the entry will be added to. If it exists the action will be appended to that group, if not a new one will be added.

5.7.4 Removing Event Listeners from Removed Nodes

When deleting elements from the model the log was sometimes filled up with `NullPointerExceptions`. These were thrown when the property view wanted to un-register from listening to their changes. The reason was that the command for deleting was executed first and after that the property view was updated. When trying to un-register the `DocumentManager` tries to get the document the element belongs to, which fails, since the element was detached when deleted. At this point there was no way of getting the event-manager-id needed for un-registering. Of course not correctly un-registering would not cause any big problems other than the `HashMap` used for managing the subscriptions would slowly fill up with unneeded subscriptions.

The solution was to copy the event-manager-id to the detached element and change the way of getting the event-manager-id from not using `getDocument` anymore and to walk up of the top-most element and there look for the id.

5.8 Integration with SimQPN

In order to integrate the SimQPN Simulator in the QPE application it was necessary to create a plug-in wrapping the call to the simulator itself. Even if it might have been possible to integrate it directly into the application this approach had one major benefit, since it allows us to distribute the editor and the simulator separately.

Basically the Simulator consists of two parts. A wizard used for gathering additional information needed to run a simulation and which is not editable using the QPE application. This additional information is stored in the original document model using meta-attributes. In its current state the wizard consists of three pages. The first is used to set the general operation mode. There are three operation modes:

- Batch Means: In this mode the simulator operates in normal run-mode and analysis method is set to “batch-means”
- Replication/Deletion: This scenario runs in normal mode but uses replication-deletion as analysis method.
- Welch: Here the analysis method is the Method of Welch.

Both Scenario 2 and 3 both need an additional parameter specifying the number of test-runs each test will have to perform.

On the second page global test-run settings are edited and finally on the last page some additional settings for places can be edited, which are not editable from within the QPE application.

Once the user has entered all needed information the QPE document and simulation configuration parameters are passed to the SimQPN simulator. The internal simulator model is built up using the QPE document and a set of XPath queries on this document.

In order to make the simulator work with as little modifications to the original version, the QPE application offers a console-view, to which Stdout is redirected. This can be used by any tool extending QPE.

6 Extending QPE

6.1 Some general information about Tables and TableViews

In order to fully understand how to extend the tables it is important to understand the TableViewer concept.

In SWT Tables are implemented by the Table class. If used without any additions adding an object of this type results in adding a simple table to the parent composite.

There are multiple things that the user might want to control:

- size of the columns
- background and foreground color of the cells
- make the cells editable

These are the options used in the QPE application. In addition a developer might be interested in cells not displaying text but checkboxes for Boolean values or even images, but we won't go into detail with these extensions.

After having created a Table object columns are defined using the following code fragment:

```
TableColumn col = new TableColumn(colorTable, SWT.LEFT);  
col.setText("Ranking");  
col.setWidth(40);
```

This creates a new column for the table “colorTable” whose content will be aligned on the left-hand side. After that the title of the column is set to “Ranking” and in the last row the width of the column in pixel is set. In most classes containing tables this is done in a method called `initTableColumns`.

Ok this is where it stops being easy. Developers wanting to change the visual appearance of a table or want to make table cells editable will be stuck here until starting to use a TableViewer. This can be thought of as the TableViewer taking full control of the table's presentation and the original table is only used as storage structure.

After initializing the table and its columns, the next step is to initialize the Table viewer. This is done by the relatively simple following code fragment:

```
colorTableView = new TableView(colorTable);
colorTableView.setColumnProperties(columnNames);
```

The first line creates the Table viewer and the second sets an array of strings containing names or ids the viewer will use to internally name the columns. This array therefore has to have as many entries as columns.

After initializing the TableView itself the content providers, label providers and cell modifiers are configured. The content provider is used for returning the content rows:

```
colorTableView.setContentProvider(new IStructuredContentProvider() {
    public Object[] getElements(Object inputElements) {
        List l = (List) inputElements;
        return l.toArray();
    }
    public void dispose() {
    }
    public void inputChanged(Viewer, Object, Object) {
    }
});
```

The only interesting method here is `getElements` which returns an array as big as the table has rows and returns an object-array containing these entries. When modifying any table, there is probably no need to modify this method. We just added it to make this tutorial complete.

After initializing the content provider the label provider is setup. It's the job of the label provider to deal with the presentation of the table in normal mode.

```
colorTableView.setLabelProvider(new ITableLabelColorProvider() {
    public void dispose() {}
    public Image getColumnImage(Object element, int columnIndex) {
        return null;
    }
    public String getColumnText(Object element, int columnIndex) {
        return null;
    }
    public void addListener(ILabelProviderListener listener) {}
    public void removeListener(ILabelProviderListener listener) {}
    public boolean isLabelProperty(Object element, String property) {
        return false;
    }
    public Color getForeground(Object element, int columnIndex) {
        return null;
    }
    public Color getBackground(Object element, int columnIndex) {
        return null;
    }
});
```



```
});
```

We stripped out the content of all methods, or the size of the listing would have been too great.

`getColumnImage` is used to return an image to be displayed in the column with the given index in the row containing the element provided as first parameter. The same way `getColumnText` returns the text that is displayed in the given cell. In the above example a custom interface is used which simply combines `ITableLabelProvider` and `ITableColorProvider`. The first interface defines the first six methods while implementing the `ITableColorProvider` interface makes the label provider able to set background and foreground color of the table.

After initializing the label provider, the cell modifier is initialized this is used to provide input for and to persist the changes made by cell editors.

```
colorTableView.setCellModifier(new ICellModifier() {
    public boolean canModify(Object element, String property) {
        return true;
    }
    public Object getValue(Object element, String property) {
        // Get the index first.
        int index = -1;
        for (int i = 0; i < columnNames.length; i++) {
            if (columnNames[i].equals(property)) {
                index = i;
                break;
            }
        }
        //////////////////////////////////
        // SNIP
        //////////////////////////////////
        return null;
    }
    public void modify(Object element, String property, Object value) {}
});
```

The first method is used to tell if the selected cell is allowed to be edited. The second returns the input fed into the cell editor for the column identified by the property parameter. In contrast to the label provider the cell modifier uses property names instead of columns. These names are the names set using the `setColumnProperties` method. Using the for-loop this name is transferred into an integer which is the same as the label provider would have provided. Finally the modify method has an additional parameter “value”. It is the new value returned by the cell editor after finishing editing.

Now the table viewer is finished initializing and as a last step the cell editors have to be initialized:

```
CellEditor[] cellEditors = new CellEditor[4];
cellEditors[0] = new ColorCellEditor(colorTableViewer.getTable());
cellEditors[1] = new TextCellEditor(colorTableViewer.getTable());
cellEditors[2] = new DoubleCellEditor(colorTableViewer.getTable());
cellEditors[3] = new IntegerCellEditor(colorTableViewer.getTable());
colorTableViewer.setCellEditors(cellEditors);
```

At first an array is defined that has as many fields as columns in the table and every array cell is initialized with a cell editor of a type suitable for editing the cell. After calling `setCellEditors` initializing the table and table viewer are ready to run.

6.2 Some general information about Trees and TreeViewers

Trees and tree viewers have a relatively closely related architecture to tables. In the following listing is shown that the general structure is identical. One thing to mention would be that this tree is used in the SimQPN wizard for configuring the place settings. This is no normal tree but a tree table combination. The first column contains a tree and the following columns behave almost like an ordinary table except the fact that in order to edit a cell in such a tree table it is necessary to first select the corresponding tree element and then the cell in the same row that is to be modified. After this code sample the differences are explained.

```
placeTree = new Tree(container, SWT.BORDER | SWT.H_SCROLL | SWT.V_SCROLL);
// Initialize the column with for column i
TreeColumn column = new TreeColumn(placeTree, SWT.LEFT, i);
column.setWidth(130);
placeTreeView = new TreeViewer(placeTree);
placeTreeView.setContentProvider(new ITreeContentProvider() { ... })
placeTreeView.setLabelProvider(new ITableLabelProvider() { ... })
placeTreeView.setCellModifier(new ICellModifier() { ... })
placeTreeView.setColumnProperties(new String[] { "Name", "statsLevel", "minObsrv", "maxObsrv" });
cellEditors = new CellEditor[] { null,
    new IntegerCellEditor(placeTreeView.getTree()),
    new IntegerCellEditor(placeTreeView.getTree()),
    new IntegerCellEditor(placeTreeView.getTree())
};
placeTreeView.setCellEditors(cellEditors);
```

The content provider no longer is an `IStructuredContentProvider` but an `ITreeContentProvider`. This has the following methods

```
public Object[] getChildren(Object parentElement)
public Object getParent(Object element)
public boolean hasChildren(Object parentElement)
public Object[] getElements(Object inputElement)
public void dispose()
public void inputChanged(Viewer viewer, Object oldInput, Object newInput)
```

Most of these methods purposes should be obvious; the only one we would like to mention in special is the `getElements` method. This returns an object array containing the content elements for the row containing the `inputElement`. It has the same job as the `getElements` method of the `IStructuredContentProvider`.

6.3 Extending Menu, Toolbar and Context Menus

Depending on which part wants to extend the menus, this has to be done in a different place. When extending it programmatically this is done in the `ActionBarAdvisor` classes. Here at first the `Action` is instantiated and registered in the `makeActions` method. After that it can be added to an existing menu or to a totally new one. Here a code sample initializing the main file menu of the application.

```
protected void fillMenuBar(IMenuManager menuBar) {
    MenuManager fileMenu = new MenuManager("&File", "qpeFile");
    menuBar.add(fileMenu);
    fileMenu.add(new Separator("new"));
    fileMenu.add(new Action());
    fileMenu.add(new Separator("open"));
    fileMenu.add(openAction);
    fileMenu.add(new Separator("close"));
    fileMenu.add(closeAction);
    fileMenu.add(closeAllAction);
    fileMenu.add(new Separator("save"));
    fileMenu.add(saveAction);
    fileMenu.add(saveAsAction);
    fileMenu.add(saveAllAction);
    fileMenu.add(new Separator("quit"));
    fileMenu.add(exitAction);
}
```

In the first row a new `MenuManager` is added to the root menu manager. This creates a new menu or submenu with the first parameter as name (and the ampersand as accelerator key) and the second parameter used as id. This id is especially important if it is intended to add content to the menu from another place in the application.

6.4 Adding additional net elements

In order to support additional net elements the following steps have to be performed:

1. An icon should be created for this new element and located in the projects icons folder.
2. Then in the package `de.tud.cs.qpe.editors.???gef.palette.templates` a class extending `DefaultElement` should be created in which name and attributes are set to their default values.
3. The proper `NetEditorPaletteFactory` in the package `de.tud.cs.qpe.editors.???gef.palette` should be extended to add a new tool entry. Either in a new drawer or in an existing one.
4. In the package `de.tud.cs.qpe.editors.???gef.property` a property composite has to be created for the new element.
5. Then the fitting `PropertyPage` in the same package has to be modified to include the new property composite in the page stack and automatically update its state.
6. Eventually the commands in the package `de.tud.cs.qpe.editors.???controller.command` have to be updated, but this depends on the type of element being added. If simply adding a new place or transition this should not be needed.
7. An `EditPart` has to be created or extended to operate the new element.
8. When adding the element to the net editor it might be necessary to update the `NetTreeEditPart` to include the element in the outline view.
9. A figure has to be created which is the graphic representation of the new element
10. Eventually additional connection anchors have to be provided, but that depends on the shape of the figure being added. Circular, rectangular and diamond shaped anchors are already available and can be reused.

6.5 Adding or changing Nodes attributes

When adding, removing or changing the attributes of a selected node only the corresponding property composite in the `de.tud.cs.qpe.editors.???gef.property` package has to be modified. Depending on the type of attribute this has to be done differently. When adding attributes directly affecting the element itself this is done in the property composites `initProperties` and `updatePropertyFields` methods. When changing tabled attributes this is a little more complicated, as the following steps have to be performed:

1. Update the `columnNames` in the constructor.

2. After that, the column definitions in the method `initTableColumns` have to be updated.
3. When `initLabelProvider` is adapted the attribute is already displayed.
4. After editing `initCellModifier` and `initCellEditors` the modifications are complete.

6.6 Extending the introduction and help system

The introduction system which also is used as help system is updated using the files in the projects “content” directory. They are simple XHTML files which are displayed in Eclipses minimal web browser used to display the pages.

6.7 Extending or changing the SimQPN wizard

Extending or modifying the wizard is not that complicated. So we will concentrate on mentioning some noteworthy parts:

The wizard pages are initialized in the `RunSimulationWizard` class located in the simulator plug-ins `de.tud.cs.simqpn.plugin.wizard` package. In the `addPages` method all pages are added to the wizard, so in order to add new pages they have to be added here.

In the wizard pages themselves the method `setErrorMessage` can be used to display an error message underneath the pages title, if set to null the error message disappears. The method `setPageComplete` is used to control the pages “next” button. If it is set to true the button is enabled, if not it is disabled. If all pages in the editor have activated Next buttons the “Finish” button is activated as well.

7 QPE Tool User's Guide

The QPE Application user interface is divided into multiple segments: an outline-view, a property-view, a console-view and a problem- and task-view and of course the main Editor view, which contains the currently active editor itself. In this introductory chapter, every individual view's general purpose is explained. After that the process of defining a new net is discussed.

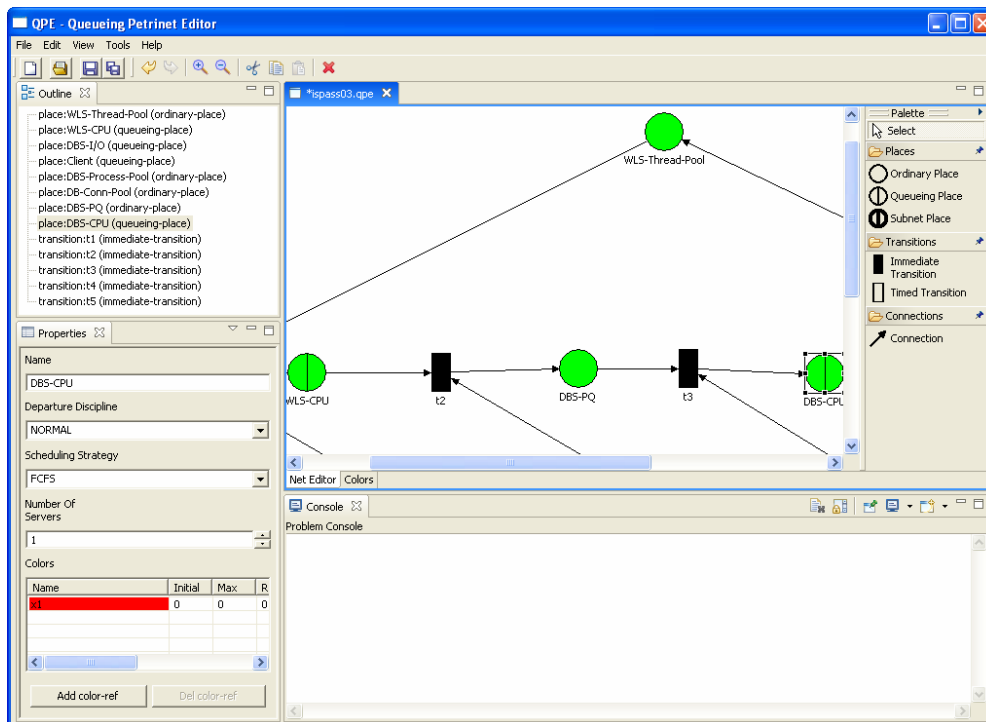


Image 23 Application overview

7.1.1 Outline View

The outline view provides a relatively primitive view of the currently active editor's content. Its purpose is to list up the resources in a table like manner for making searching for an individual component easier than when looking through the graphical editor. Elements selected in this view are focused in the graphical editor and the editor scrolls so the selected part is visible, if the selected part was currently not visible.

7.1.2 Property View

The property view enables the user to manipulate the properties of the currently selected Element. The content of this view depends on the type of selected node. Basically it consists of direct data such as name of the element and general properties such as departure discipline in places and a table containing color-ref information when dealing with places or mode

definitions with transitions. Depending on the type of place or transitions the number and type of global and table properties may vary. The main concept however is the same.

In order to add a mode or color-ref to an element, the add-button underneath the table is used. When defining color-refs for places you have to keep in mind that this button is deactivated if every color defined for the net has already been added to the current place. One of the most common reasons for this will probably be that no colors have been defined in a new net. In order to add additional color-ref you have to create new colors first.

Each color-ref or mode has multiple child attributes. These are edited by clicking inside the table. If the value is editable an appropriate editor will let the user edit the value in-place. If an invalid value was entered when loosing focus the original value will automatically be restored and no change is made to the document itself. If a value is valid, the document is modified which will instantly result in the dirty-state of the document being changed, which is signaled by a small asterisk in the editors tab.

7.1.3 Console View

The console view has no major purpose when using the editor without any extensions such as the SimQPN plug-in. Its purpose is to present extensions output in one central view without the user having to monitor log files simultaneously. It contains some controls for clearing the view, for locking it (This is particularly useful when looking at a certain part of the log). Additional log entries are still appended to the console, but the focus is locked to the current position.

7.1.4 Problem View

The problem view is used to report any problems in the currently opened document. It consists mainly of a plain table containing one row per error or warning the RuleEngine could find. This table is updated every time a document is modified. Since the RuleEngine can load additional rule-sets this allows to report errors when using the SimQPN simulator that would not have been reported without it.

7.2 Editing Nets

When creating a new net the user is presented an empty net editor. This Editor has two Views: One for editing the net and one for editing its colors. It makes sense to start defining the colors or it will not be possible to add color-refs to any place and this step would have to be done after defining some colors.

7.2.1 Defining Colors

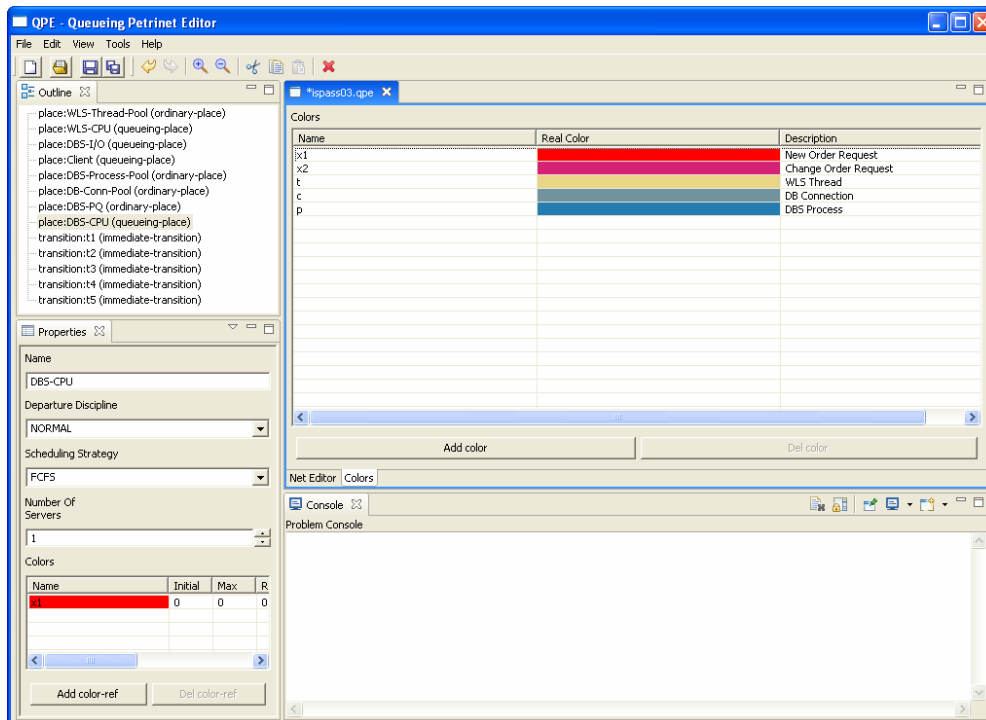


Image 24 Color editor

When selecting the “Colors” Tab at the bottom of the net editor the above screen will appear. Its interface is relatively simple. It consists of a Table showing all colors already defined for the net and two buttons for adding and deleting colors. The delete button is only active if a color is selected. When clicking on the add button a new entry will be added at the end of the table which is predefined with the name “new color” and if adding multiple colors they will be suffixed with a rising number “new color 1”, “new color 2” and so on. When clicking inside the table field the field becomes editable and the user can change the colors name.

Besides the name, each color has a “real-color” field used to be able to distinguish the colors better. When clicking inside this field a color-chooser dialog appears in which the user can choose a color. The third field has no real effect in the editor itself but can be used to note what the color is used for. In the above example nobody could really guess what x1 is used for but when having a look at the color table you can see that this is a “New Order Request”.

7.2.2 Defining Nets

After some colors have been defined the definition of the net can be started. For this the “Net Editor” tab has to be selected at the bottom of the editor area.

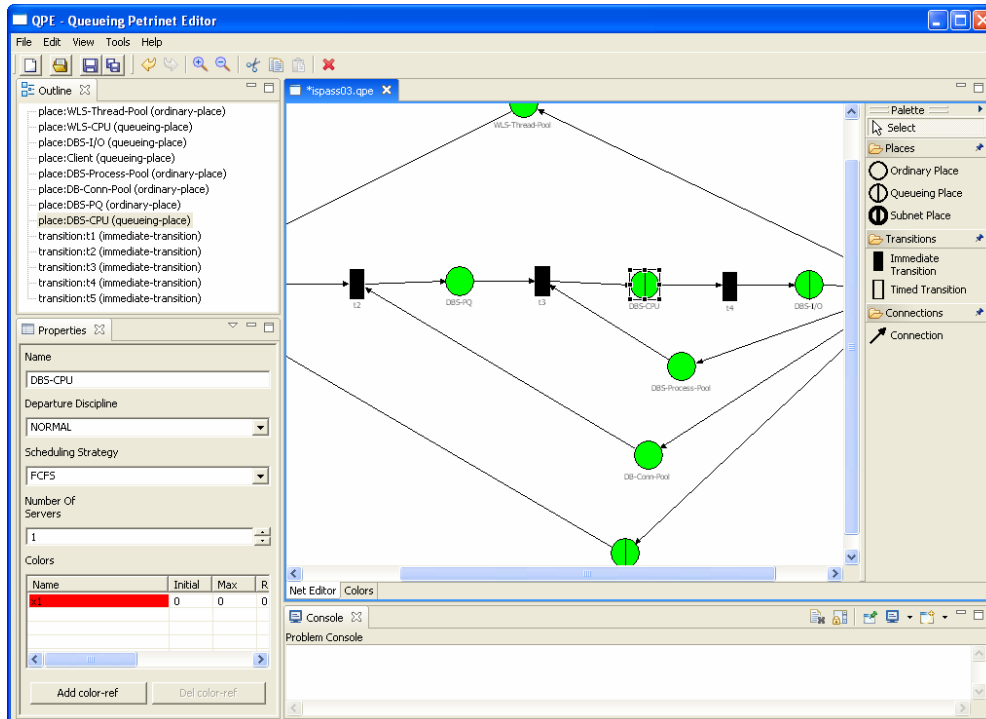


Image 25 Net editor

On the right-hand side of the editor the fly-out-palette is located. In its initial configuration it is represented by a thin bar. As soon as the user places the mouse pointer over this bar the palette unfolds to its full size. If the user wants to stay unfolded, the small arrow-symbol at the top of the bar can be used. One in fixed mode the user can reactivate the dynamic mode by re-clicking on the arrow symbol.

The palette is built up of a set of so-called drawers, which are initially all open, which contain multiple tools. By double-clicking on the drawer title the user can open and close individual drawers.

In order to place a net element in the net, the user has to select the type of item he wants to create and click inside the net editor area. Every tool except to topmost one – the selection tool – is used to create net-elements.

The connection tool however is a special case, since connections aren't added to the net itself but to places or transitions which are members of the net.

When creating a connection at first a starting element has to be selected. If the element underneath the mouse is a valid element, the symbol of a no-entry-sign disappears. After connecting the starting point of a connection the target has to be selected to finally add the connection.

There are a few constraints to connections and the connection is only allowed to be attached to the target if these constraints are all met. A connection has to go from a place to a transition or from a transition to a place. Connections between places or transitions are not allowed. Then

only one connection is allowed from one node to another in a direction. If a connection from node A to node B exists, no further connection from A to B is allowed, but if no connection from B to A exists, then the user is allowed to add that connection.

A connection between a place and a transition makes all color-refs defined for that place available in the incidence function of the transition. Depending on if the connection points to or from the transition the color-refs are displayed as input-colors or output-colors. Incoming connections to the transition add input colors and outgoing connections add output colors.

A connection can be reconnected by selecting it and clicking on the small black box at the connections end and simply dragging it to its new destination. Remember that when performing this action the consistency checks are performed as if the connection was a new one. The source or target can only be connected to valid destinations.

A place has to possess at least one color-ref and a mode at least one mode. Since places and transitions initially contain none of them they have to be defined before being able to do something with the elements. This is done using the property view of the places and transitions. After selecting a place the property view shows a table with all color-refs defined for the current place. If there are unreferenced colors left in the net, the “Add Color-Ref” button is active. Clicking on it adds the first unused color to the element. If this was not the last free color (the add button is still active) by clicking on the first column of the table a drop down list allows the user to select an unused color which will be referenced by this color-ref.

This is basically all that is needed to define color-refs. Of course the user has to adjust the remaining properties to fit his needs, but this can be done afterwards.

Adding Modes is a little simpler, since they are defined per transition instead of per net and therefore the “Add Mode” button is always available. Instead of the drop-down list for choosing there are two fields for editing name and real-color of the mode.

Apart from the palette there are a number of tools available for modifying the net. When opening the first net editor an additional view-menu appears in the menu bar. This contains tools for undoing or redoing actions, for cut, copy, paste, deleting and selecting all elements of the net.

In addition to these standard options there are menu entries for zooming. This feature is especially useful when dealing with large nets. The same functionality can be achieved by pressing the CTRL key and using the scroll-wheel of the mouse.

Even if the main functionality of copy and paste might be obvious to a normal user, in this case there is a little more to them as in most other editors. Basically there is a great difference between pasting inside the same net or in a completely different one.

When pasting to the same net, the content was copied from, the selected nodes are simply duplicated and inserted again with a small offset so no elements are hidden by others.

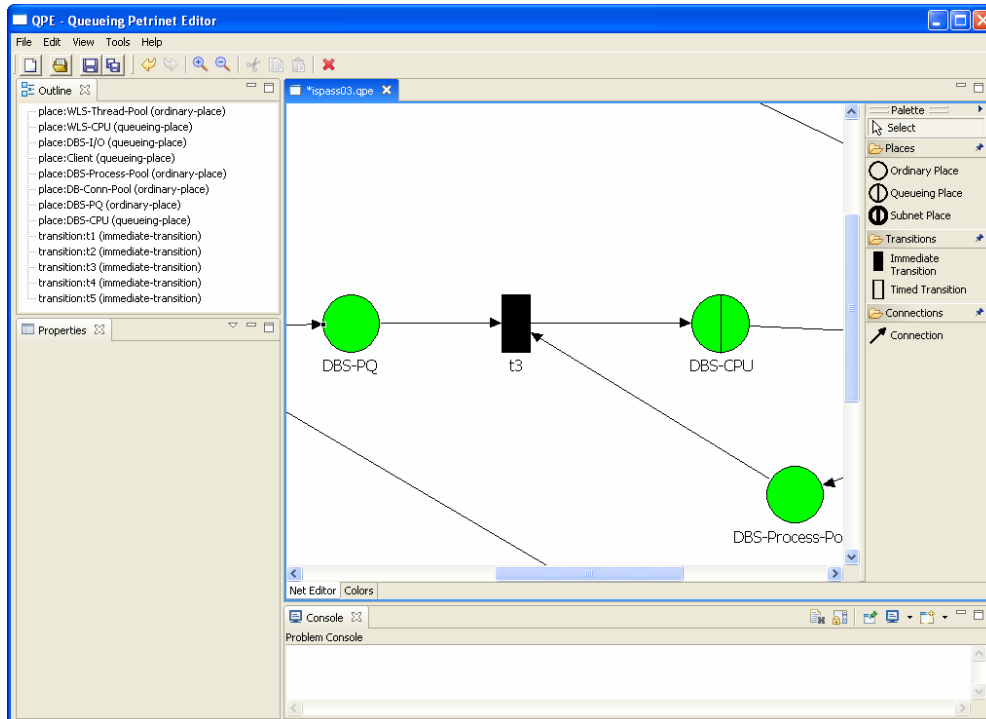


Image 26 Net prior to paste operation

Any connection within the selection is duplicated, so if a connection exists from A to B and both elements were copied, then after pasting there is a connection from A to B and one from the copy of A to the copy of B. Any connection between a selected and a non selected node will be duplicated. If there was a connection from A to B and B was duplicated, then in the resulting net there is a connection between A and B as well as between A and copy of B. The incidence functions of the copies are identical to the ones of the original elements.

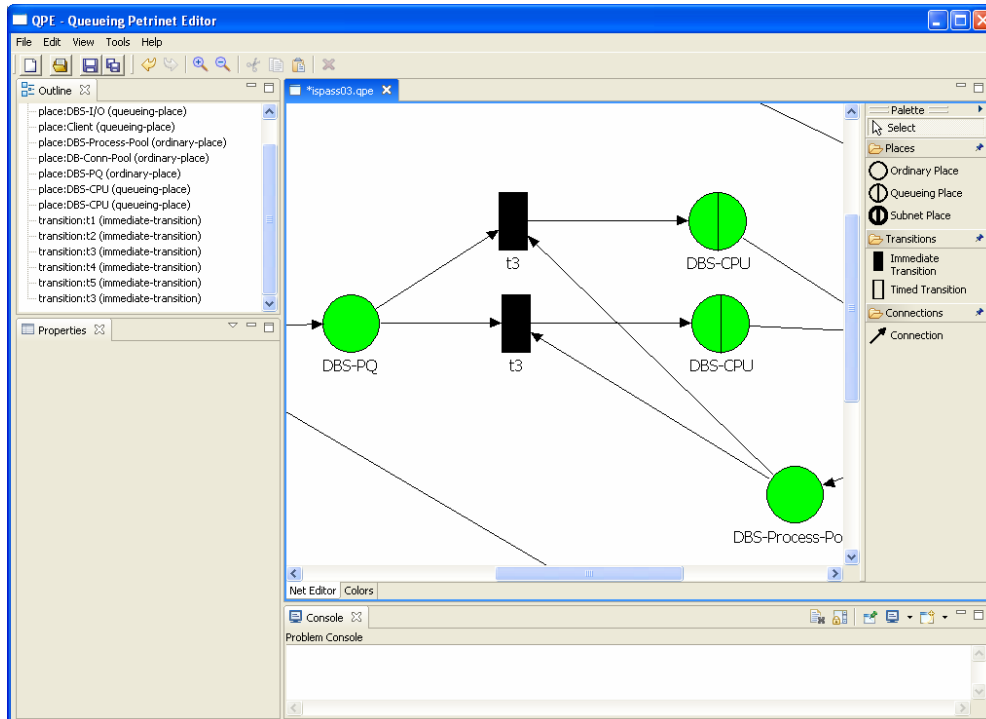


Image 27 Net after pasting to same document

When copying from one document to another everything is a lot more complicated. At first colors existing in the source document generally don't exist in the target document. This is why the color-definitions referenced in the selections places are pasted into the target document and their names are prefixed with the name of the source document.

The next difference is that connections to and from the selection to the rest of the source document don't exist in the target document. This is why these connections are stripped from the selection when being inserted. It is obvious that this results in further problems. Any transition losing a connection by this has to be cleared of connections in the incidence function that reference color-refs accessed through that connection. The result of pasting to a new document is shown in the following screenshots of the resulting net and color editor.

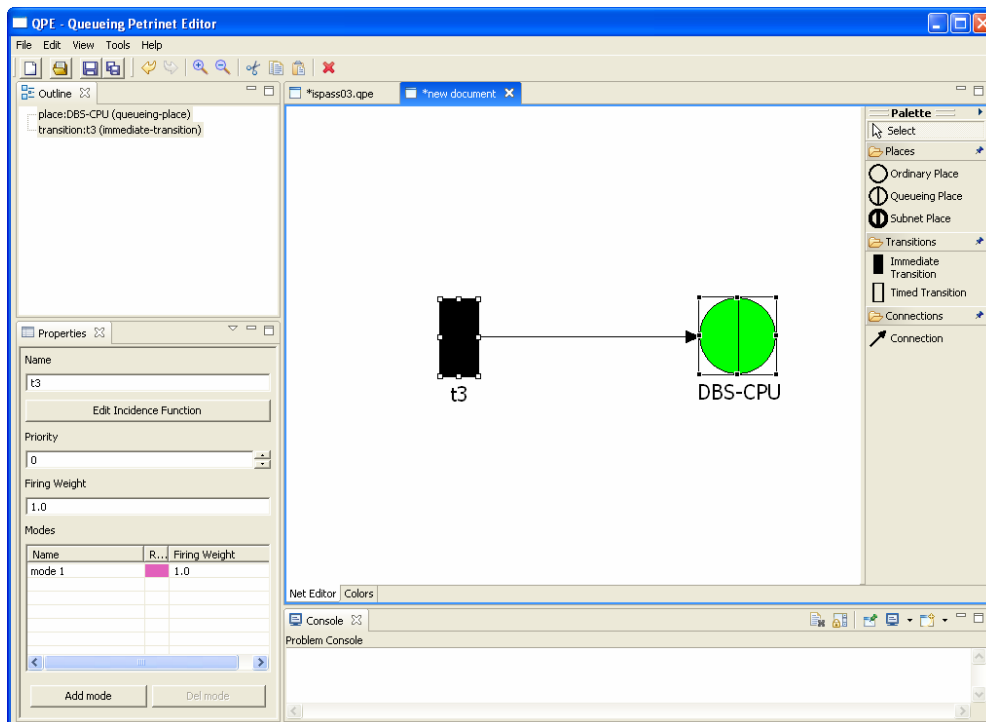


Image 28 Net after pasting to new document

The screenshot shows the QPE - Queueing Petrinet Editor interface with the 'Colors' panel open. The panel contains a table with the following data:

Name	Real Color	Description
jspass03.x1		New Order Request

The 'Properties' panel for the selected transition 't3' shows the same settings as in Image 28. The 'Colors' panel also includes 'Add color' and 'Del color' buttons at the bottom.

Image 29 Colors after pasting to new document

7.3 Editing Incidence functions

After defining color-refs and modes for all places and transitions the incidence functions can be defined. For doing this a transition has to be selected. The incidence function editor can be opened by simply double-clicking on the transition or by clicking on the “Edit Incidence Function” button of the selected transitions property view. Both result in opening the incidence function editor for the transition.

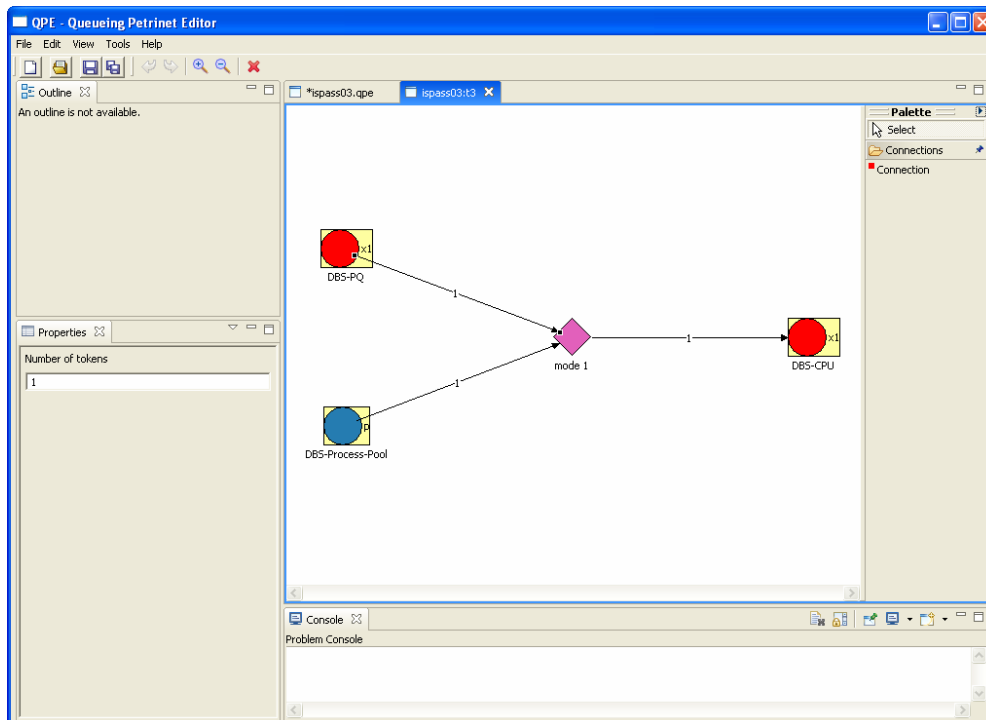


Image 30 Incidence function editor

The main structure of the editor is very similar to the net editor's layout. It too has a palette which can fly out or be docked the same way as in the net editor. The property view is used to modify the attributes of a selected object. In case of the incidence function editor, the only elements that can be added or modified are connections. This is why the palette only contains the selection and connection tools.

The layout of the editor is fixed. On the left-hand side for every incoming connection there is a place figure with all color-refs defined for that place. If a place possesses more color-refs it is bigger, if it only has few it is smaller. In the middle the modes are located, which were defined for that particular transition. On the left-hand side the places are located to which outgoing connections in the net editor exist. A place can be shown twice in this view, if an incoming connection from that place and an outgoing connection to that place exist.

The only things the user can edit in this view are the connections. Connections can only be made from color-refs to modes or from modes to color-refs. To be a little more specific,

connections are only allowed from input places color-refs to modes or from modes to output places color-refs and only one connection is allowed from one node to another.

In contrast to connections in the net editor, connections in the incidence function contain a label indicating the number of tokens used for firing that mode it is connected to. It is editable using the text-field in the property view when the connection is selected.

Connections can be removed or reconnected the same way as in the net editor. Only the context menu and view menu contain a lot less options since copy and paste makes no sense in the incidence function.

7.4 Running SimQPN

The SimQPN plug-in adds an entry to the QPE tools menu. A document has to be opened in order to activate the plug-in. When activated the user is presented a wizard for configuring the test-run prior to starting the simulation-run.

7.4.1 Select Run Configuration

The first page of the wizard is used for managing the configurations. A configuration is a named set of settings for the simulator. This functionality was added after finishing a first version of the wizard because it became obvious that having to reconfigure the entire net after changing the simulation-mode was very user unfriendly.

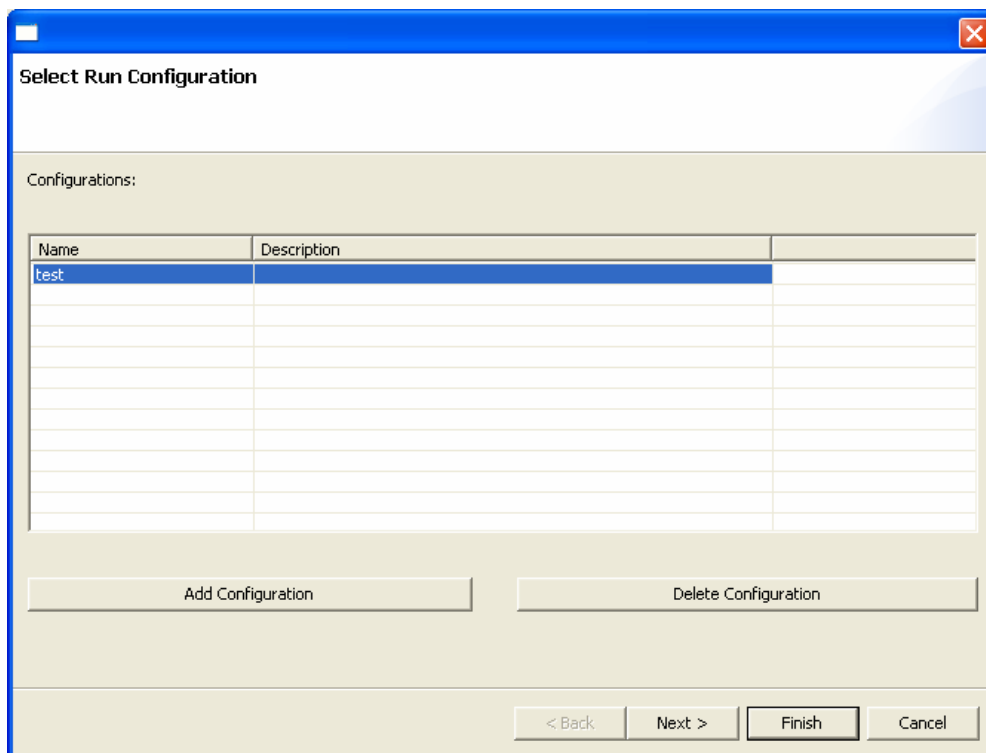


Image 31 Configuration page

To add a new configuration the user has to press the “Add Configuration” button. After the new configuration has been added, the user can modify the name and description of it by simply clicking on the elements the user wants to modify.

In order to be able to delete a configuration a configuration has to be activated by clicking somewhere in the desired configurations row. The “Delete Configuration” button is then activated. As soon as a configuration is selected the “Next” button at the bottom of the wizard is activated and the user can start configuring the selected configuration.

7.4.2 Select Analysis Method

On the next page the general mode of the simulator has to be selected. In its current version three modes or scenarios are supported by the plug-in:

- Batch Means
- Replication/Deletion
- Method of Welch

Batch Means takes no additional parameters, but Replication/Deletion and Method of Welch need an additional “number of runs” parameter. As soon as one of these methods is selected, the corresponding text field is activated and it is initialized with a default value. As soon as the user changes this to an invalid value an error message is displayed in the area underneath the wizard page title and the “Next” button is deactivated. The user can only continue after changing the value to a valid one.

Select Analysis Method

☒ Batch Means Steady state analysis using the method of non-overlapping batch means.

☐ Replication/Deletion Steady state analysis using the replication/deletion approach.

Number of runs

☐ Method of Welch Analysis of the length of the initial transient (warm-up period) using the method of Welch.

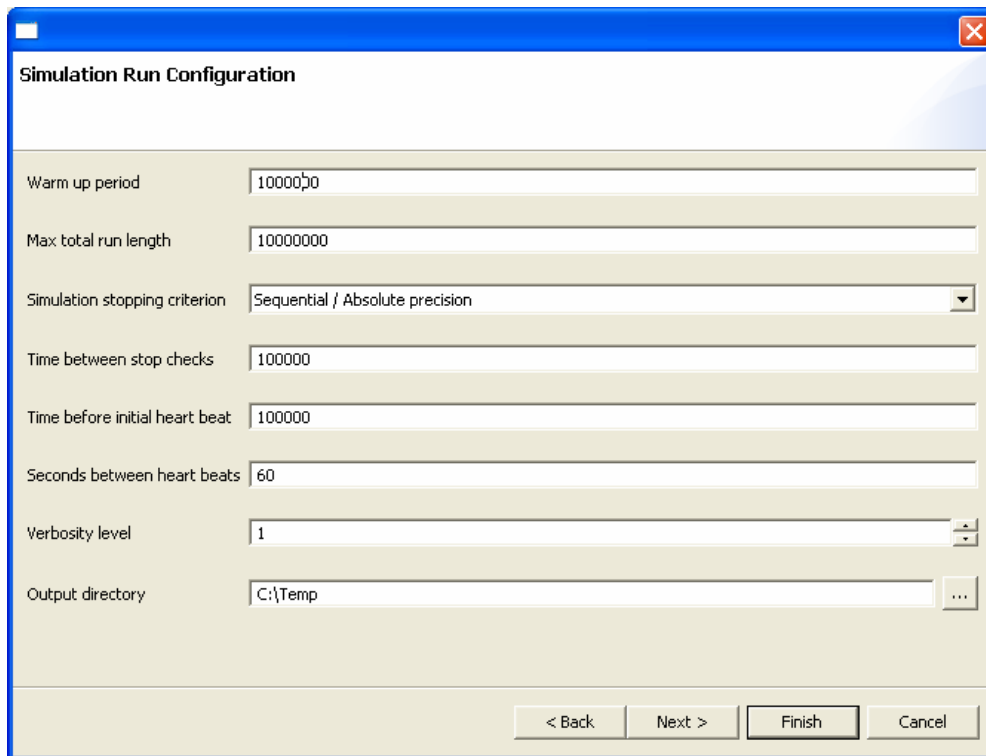
Number of runs

< Back Next > Finish Cancel

Image 32 Analysis method page

7.4.3 Simulation Run Configuration

On this page all mode specific global parameters have to be specified. Depending on the selected mode some fields may not be editable or even be hidden. The screenshot shows a typical Batch Means screen. The field “Simulation Stopping Criterion” is usually set to “Fixed sample rate” which usually hides the “Time between stop checks” field, since it is only used for “Sequential / Absolute Precision” and “Sequential / Relative Precision”. On this page invalid values are reported the same way as on the mode-selection page. As soon as all values are set to valid ones the “Next” button is activated. This should be deactivated by default, because no valid directory has been set. After setting this value using the Directory-chooser dialog opening when pressing on the button behind the text field, the user can proceed to the next and final wizard page.



The image shows a 'Simulation Run Configuration' dialog box with a blue title bar and a standard Windows-style window. It contains several input fields and a dropdown menu. The fields are: 'Warm up period' (1000000), 'Max total run length' (10000000), 'Simulation stopping criterion' (Sequential / Absolute precision), 'Time between stop checks' (100000), 'Time before initial heart beat' (100000), 'Seconds between heart beats' (60), 'Verbosity level' (1), and 'Output directory' (C:\Temp). At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Warm up period	1000000
Max total run length	10000000
Simulation stopping criterion	Sequential / Absolute precision
Time between stop checks	100000
Time before initial heart beat	100000
Seconds between heart beats	60
Verbosity level	1
Output directory	C:\Temp

Image 33 Simulation run configuration page

The fourth and final wizard-page is only available in “Bach Means” and “Method of Welch” mode and is used for configuring additional parameters for ordinary and queueing places. In the first column of the table the places are listed. For every place the user can specify a “statsLevel” parameter. It defaults to 1 and can be set to values from one to four. If set to level three and four the places rows can be opened like in an ordinary tree. On this level all color-refs of the place are listed.

If the place is an ordinary place there is one row per color-ref containing all parameters for it in the current table row. It is important to note that the current implementation of the tree-table requires the user to select an element of the tree and then can edit the table cells belonging to that row. We hope in future versions this will be changed.

If the place was a queueing place there is another level in the tree. There still is one row per color-ref, but this is not editable. These rows are expandable the same way the places were. When opening the next step branch of the tree there are two rows per color-ref: One with settings of the depository and one for the queue. The behavior is when editing is identical to that of ordinary places.

If an invalid value is entered in one of the field, then it will be reset to its old value and a message describing the correct format will be shown in the default location underneath the wizard pages title.

Configuration Parameters for Batch Means Method

Configure simulator-specific place parameters.

Place Settings

Name	statsLevel	signLev	reqAbsPrc	reqRelPrc	batchSize	minBatches	numBMea...
WLS-Thread-Pool	1						
WLS-CPU	3						
x1							
depository		0.05	50	0.05	200	60	50
queue		0.05	50	0.05	200	60	50
DBS-I/O	1						
Client	3						
x1							
depository		0.05	50	0.05	200	60	50
queue		0.05	50	0.05	200	60	50
DBS-Process-Pool	1						
DB-Conn-Pool	1						
DBS-PQ	1						
DBS-CPU	1						
DBS-CPU	1						

< Back

Next >

Finish

Cancel

Image 34 Place configuration page

8 Conclusion

At first it was important getting an overview over QPNs. After that, tools for editing and analyzing as well as data formats for describing and storing QPNs were evaluated.

The results were used to work out the differences between the different approaches. At first work was concentrated on defining an extensible data format. At first based upon PNML as being a quasi standard for Petri nets, creating a PNML Schema usable for describing QPNs seemed the best approach. Unfortunately PNML was designed universally that it supported many aspects that were not needed or would not supported by the application we were going to develop, aspects such as:

- multiple nets in one document or
- pages with parts of nets which connections between the pages

On the other hand it would have been impossible to add support for colors, defined on a per-net basis. It was therefore decided to throw direct PNML support over board and work with a format especially designed from scratch for supporting QPNs. This resulted in the QPN format described in chapter 3. Omitting direct PNML support however does not necessarily mean that it is impossible to work with PNML input. Everything needed for this would be an import mechanism based on two XSL transformations: One for importing and one for exporting.

Parallel to developing the data format evaluation work for the application was started. HiQPN was the only tool supporting all desired features. Unfortunately it was only was able to run on the Solaris platform. All other tools were missing great portions of functionality.

Work was started on extending the PIPE editor since it seemed to be best suitable for being extended to the needs of QPNs. Unfortunately this was a misjudgment. Extending an editor for editing incidence functions would have meant either patching the code-base or almost totally reimplementing the PIPE core. Since the general structure of the application was not that good it was decided to stop working on the PIPE editor and we started working on a version designed from scratch using visualization frameworks for dealing with the visualization.

After having a brief look at the Touchgraph library work was started on an Eclipse and GEF based editor. After first successes it turned out that Eclipse is missing greatly in good documentation. There is no birds-eye-view of the platform. There is no map helping new developers find their way. Many days and even weeks were lost because of the lack of information. Nevertheless we were able to sort out most of the problems and the result is a graphical modeling and simulation environment suitable for creating and analyzing large QPN models.

9 Future work

After finishing this thesis work on the QPE application is not yet finished the following tasks still have to be dealt with:

1. Mapping of QPE's internal data format to the simulators data structures has to be finished, since some information for finishing has still to be provided.
2. Validation of the simulator integration.
3. Implementation of a subnet editor.
4. Implementation of a custom problem view for reporting problems in the net.
5. Renaming and branding of QPE to be able to run as standalone application on all supported platforms.

10Bibliography

SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation

Samuel Kounev, Alejandro Buchmann

Performance Evaluation, Vol. 63, No. 4-5, Elsevier Science, May 2006

Performance Modelling of Distributed E-Business Applications using Queueing Petri Nets

Samuel Kounev, Alejandro Buchmann

Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software

(ISPASS'03), March 2003

A Methodology for Performance Modeling of Distributed Component-Based Systems using Queueing Petri Nets

Samuel Kounev, In review. IEEE Transactions on Software Engineering, 2005

Building and delivering a table editor with SWT/JFace

Laurent Gauthier, Eclipse Website, 2003,

http://www.eclipse.org/articles/Article-Table-viewer/table_viewer.html

Eclipse Platform Technical Overview

Wayne Beaton, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html>

Eclipse Workbench: Using the Selection Service

Marc R. Hoffmann, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html>

Rich Client Tutorial Part 1,

Ed Burnette, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html>

Rich Client Tutorial Part 2

Ed Burnette, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Article-RCP-2/tutorial2.html>

Rich Client Tutorial Part 3

Ed Burnette, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Article-RCP-3/tutorial3.html>

Creating JFace Wizards

Doina Klinger, Eclipse Website, 2006,

<http://www.eclipse.org/articles/Article-JFace%20Wizards/wizardArticle.html>

Inside the Workbench: A guide to the workbench internals

Stefan Xenos, Eclipse Website, 2005,

<http://www.eclipse.org/articles/Article-UI-Workbench/workbench.html>

A Shape Diagram Editor

Bo Majewski, Eclipse Website, 2004,

<http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html>

How You've Changed!: Responding to resource changes in the Eclipse workspace

John Arthorne, Eclipse Website, 2002,

<http://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html>

EMF goes RCP

Marcelo Paternostro, Eclipse Website, 2004,

<http://www.eclipse.org/articles/Article-EMF-goes-RCP/rcp.html>

Building a Database Schema Diagram Editor with GEF

Phil Zoio, Eclipse Website, 2004,

<http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html>

Branding Your Application

Andrew Eidsness and Pascal Rapicault, Eclipse Website, 2004,
<http://www.eclipse.org/articles/Article-Branding/branding-your-application.html>

Notes on the Eclipse Plug-in Architecture

Azad Bolour, Eclipse Website, 2003,
http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

Take control of your properties

Dicky Johan, Eclipse Website, 2003,
<http://www.eclipse.org/articles/Article-Properties-View/properties-view.html>

How to use the JFace Tree Viewer

Chris Grindstaff, Eclipse Website, 2002,
<http://www.eclipse.org/articles/treeviewer-cg/TreeViewerArticle.htm>

Understanding Layouts in SWT

Carolyn MacLeod, Eclipse Website, 2001,
<http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm>

Queueing Petri Nets (QPNs)

Falko Bause, Peter Buchholz, Peter Kemper, Universität Dortmund, 2003
<http://ls4-www.informatik.uni-dortmund.de/QPN/>

HiQPN-Tool

Heiko Rölke, University of Hamburg, 2003,
<http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db/hiqpntool.html>

Design/CPN

CPN Group, University of Aarhus, 2006,
<http://www.daimi.au.dk/designCPN/>

PIPE2

James Bloom, Clare Clark, Camilla Clifford, Alex Duncan, Haroun Khan and Manos Papantoniou, Tom Barnwell, Michael Camacho, Matthew Cook, Maxim Gready, Peter Kyme and Michael Tsouchlaris, Nadeem Akharware, Imperial College London, 2005,
<http://pipe2.sourceforge.net/about.html>

Projekt PED

Gerhard H. Schildt, Vienna University of Technology, 2005,
<http://www.auto.tuwien.ac.at/~jlukasse/>

JARP

Ricardo Padilha, Sourceforge, 2005,
<http://jarp.sourceforge.net/us/index.html>

PNS - An S/T Petri-Net Simulation System

Michael Ancutici, Thomas Bräunl,
<http://robotics.ee.uwa.edu.au/pns/>

PENECACHROMOS

W. Fengler, TU-Ilmenau, 2002,
http://tin.tu-ilmenau.de/ra/skripte/pn_chr/

Renew

Doug Lea, University of Hamburg, 2004,
<http://www.renew.de/>

Eclipse 3.1 help system

Eclipse Community, 2006,
<http://help.eclipse.org/help31/index.jsp>

Display a UML Diagram using Draw2D

Daniel Lee, Eclipse Webseite, 2003,
<http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html>

Programmierung:Plugins:Code Schnipsel:Konsolenausgabe

http://www.plog4u.de/index.php/Programmierung:Plugins:Code_Schnipsel:Konsolenausgab

Wizards in Eclipse - An introduction to working with platform and custom wizards

Azra Fatima, HP, 2003

http://devresource.hp.com/drc/technical_articles/wizards/index.jsp

SWT Programming with Eclipse

Koray Guclu, Developer.com,

http://www.developer.com/java/other/article.php/10936_3330861_5

Eclipse Workbench Extension Point Action Sets

Eclipse Website

<http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.ui/doc/Attic/actionExpressions.html?rev=1.3>

Eclipse Plugin Development (Connection Anchors)

BlogJava.net, 2006,

<http://www.blogjava.net/reloadcn/archive/2005/09/07/12279.aspx>

Code Samples using Zoom Manager

http://www13.plala.or.jp/observe/GEF/GEF_Hello10.html

The Eclipse task list

Benoit Marchal, IBM, 2004

<http://www-128.ibm.com/developerworks/library/x-wxxm27/>

11Index

- action 19, 22, 24, 25, 26, 32, 33, 36, 37, 38, 40, 45, 46, 47, 60
- actual population place 6
- attributes 2, 9, 10, 11, 12, 13, 14, 25, 32, 48, 54, 57, 64
- AWT 16, 19
- background 3, 49, 51
- batch beans 48, 66, 67
- bounds 38, 39, 43, 44
- bundle 34
- CGSPN 5
- code 1, 2, 7, 16, 17, 20, 22, 26, 30, 31, 32, 33, 34, 36, 40, 49, 52, 53
- color 2, 4, 6, 9, 10, 11, 14, 16, 25, 32, 36, 39, 43, 57, 58, 60, 62
- color definitions 9
- color function 3, 4
- colored transition 2
- color-ref 9, 11, 13, 43, 44, 57, 60, 62, 64, 65, 68
- command 32, 37, 38, 41, 47, 54
- command stack 32
- configuration 10, 65, 66
- connection 3, 9, 10, 11, 12, 13, 16, 18, 19, 25, 32, 33, 36, 37, 38, 39, 41, 43, 54, 59, 60, 61, 62, 64, 65
- connection anchor 39, 43
- console view 23
- console view 25
- context menu 33
- contributor 32
- controller 33, 36, 46, 47, 54
- CPN 3, 4, 5
- creation tool 33
- debug 20, 24
- depository 5, 6
- Design/CPN 1, 7
- discrete event simulation 7
- distributed systems 1, 3
- document 9, 10, 14, 22, 23, 26, 28, 29, 30, 32, 44, 47, 48, 57, 62, 63, 65
- document type definition 14
- DocumentManager 27, 28, 29, 30, 31, 44, 47
- Dom 27, 30, 36
- Dom4J 30, 32
- Drag & Drop 19
- Eclipse iv, 2, 19, 20, 21, 22, 23, 24, 25, 26, 32, 33, 34, 35, 36, 38, 40, 45, 47
- Eclipse group 19
- Eclipse runtime 23
- e-commerce systems 3
- edit policy 32, 37, 38, 41
- editor 1, 2, 7, 8, 16, 18, 22, 23, 24, 25, 26, 31, 32, 33, 34, 36, 39, 40, 41, 42, 43, 46, 51, 52, 54, 57, 60
- EditPart 36, 37, 38, 41, 47, 54
- EditParts 26, 36, 37, 38, 41, 43
- event 28, 29, 36, 37, 40, 44
- extension 9, 10, 17, 22, 23, 26, 31, 45, 46, 47
- factory 33, 36
- figure 39, 43, 44, 47, 54, 64
- fire 3, 4, 5, 6, 28, 30, 65
- firing delay 6
- firing frequency 6
- firing weight 4, 5, 6, 12
- framework 2, 16, 18, 19, 32
- GEF 2, 19, 22, 31, 32, 33, 39, 40
- global test-run settings 48
- GSPN 4, 5
- GUI 16, 19
- handle-bounds 39
- HiQPN 1, 2, 6, 7, 8
- HQPN 6
- IDE 19, 24
- immediate queueing place 5, 6
- immediate transition 4, 5, 6, 12, 16, 43
- incidence function 2, 7, 12, 17, 32, 36, 39, 40, 41, 43, 44, 60, 62, 64, 65
- incidence function editor 2, 13, 17, 31, 32, 36, 39, 40, 41, 64
- input place 3, 6, 41, 42, 65
- interface 1, 17, 20, 29, 35, 39, 40, 51, 56, 58

JARP 1, 7
 Java 1, 16, 19, 24, 41
 lifecycle 23, 25
 Markov Chain 6
 mathematical definition 3
 mean-firing- delay 12
 menu 22, 24, 25, 26, 32, 33, 40, 45, 46, 47, 53, 60, 65
 meta-attribute 9, 10, 11, 14
 mode 2, 4, 9, 12, 25, 39, 42, 43, 44, 48, 50, 56, 57, 59, 60, 64, 65, 66, 67, 68
 model 1, 3, 7, 9, 16, 26, 27, 28, 29, 30, 31, 33, 36, 38, 41, 47, 48
 model view controller 33
 MVC 33
 net 1, 2, 3, 4, 5, 6, 9, 10, 11, 16, 17, 18, 20, 25, 30, 31, 32, 33, 38, 39, 40, 41, 44, 46, 47, 54, 56, 57, 58, 59, 60, 61, 62, 64, 65
 net editor 31, 32, 36, 37, 39, 40, 43
 net-element 9
 networks of homogeneous elements 18
 OLE 19
 open-source 16
 ordinary place 6, 12, 16, 43, 68
 outline view 25
 output place 3, 6, 41, 42, 65
 output transition 6
 palette 32, 33, 54, 59, 60, 64
 paper 2
 Peneca Chromos 1, 7
 performance 1
 perspective 22, 23, 24, 26, 35
 Petri net 1, 18
 - colored 3
 - colored generalized stochastic 5
 - conventional 3
 - generalized stochastic 4
 - hierarchically combined queueing 6
 - queueing 1, 5
 - stochastic 4
 PIPE 1, 2, 7, 16, 17, 18
 place 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 18, 19, 22, 25, 26, 32, 36, 37, 38, 39, 40, 41, 43, 44, 47, 48, 52, 53, 54, 56, 57, 59, 60, 62, 64, 68
 - immediate queueing 5
 - input 3, 6, 41, 42, 65
 - ordinary 6, 12, 16, 43, 68
 - output 3, 6, 41, 42, 65
 - queueing 3, 5, 6, 14
 - subnet 6, 12
 - timed queueing 5
 platform 34
 Platform 31, 34
 plug-in 9, 10, 16, 19, 20, 22, 23, 25, 26, 33, 34, 44, 45, 47, 55, 57, 65, 66
 plugin.xml 22, 23, 26, 31, 32, 45, 46
 PNML 2, 9
 PNS 1, 7
 POJO 26, 27, 30
 problem 45
 problem view 25, 44
 Project PED 1, 7
 proof of concept 44
 property page 35, 40
 property sheet 35
 property view 25, 34
 QUEST 2
 QPE i, iv, 7, 8, 9, 10, 14, 17, 20, 22, 23, 25, 28, 31, 33, 34, 35, 39, 43, 44, 47, 48, 49, 56, 65
 QPN 5, 6, 9, 12
 quasi standard 2
 queue 5
 queueing discipline 5
 queueing place 1, 2, 3, 5, 6, 9, 11, 14, 68
 RCP 19, 20, 22, 23, 25
 real color 10
 refactoring 2, 14, 17, 27
 relative firing frequency 5
 RelaxNG 14
 Renew 1, 7
 replication/deletion 48, 66
 resources 22, 23, 34, 56
 Rich Content Platform 19, 22, 23
 Rule Manager 44
 RuleEngine 23, 25, 28, 44, 57
 scalability 1
 scheduling 5
 schema 14
 selection tool 33
 separators 26
 SimQPN 1, 7, 22, 44, 47, 48, 52, 55, 57, 65
 simulator 1, 7, 10, 22, 47, 48, 55, 57, 65, 66
 Snoopy 1, 7

- Solaris 1, 2, 6
- sourceforge 16
- SPN 4
- state space explosion problem 1, 6, 7
- subnet place 2, 6, 12
- SWING 16, 19
- SWT 19, 20, 35, 49, 52
- template 33
- timed queueing place 5, 6
- timed transition 2, 4, 5, 6
- token 3, 4, 5, 6, 11, 12, 65
- tool bar 25, 32
- toolbar 22, 24, 25, 26, 40, 45
- Touchgraph 18
- transition 2, 3, 4, 5, 6, 7, 9, 10, 12, 13, 14, 16, 18, 19, 25, 32, 37, 38, 39, 41, 43, 44, 54, 57, 59, 60, 62, 64
 - immediate 4, 5, 6, 12, 16, 43
 - output 6
 - timed 2, 4, 5, 6
- UNIX 19
- validation 14
- validity checks 44
- view 16, 20, 22, 24, 25, 26, 33, 34, 35, 36, 38, 39, 40, 42, 43, 44, 45, 46, 47, 48, 54, 56, 57, 60, 64, 65
- visualization framework 18
- W3C Xml-Schema 14
- weight 3
- welch 48, 66, 68
- window 25, 32, 45, 46
- Windows 19
- workbench 23, 24
- XML 2, 9, 14, 19, 27, 30, 44
- XPath 14, 27, 48
- XSD 14
- XSL 2, 9, 14