

# Renew – Architecture Guide

Olaf Kummer

Frank Wienberg

Michael Duvigneau

Lawrence Cabac

Michael Haustermann

David Mosteller

Marcel Hansson

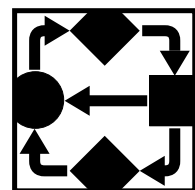
University of Hamburg

Department for Informatics

Theoretical Foundations Group

Release 5.0

March 26, 2026



This manual is ©2026 by Olaf Kummer, Frank Wienberg, Michael Duvigneau, Lawrence Cabac, Michael Haustermann, David Mosteller, Marcel Hansson.

Arbeitsbereich ART  
— Renew —  
Fachbereich Informatik  
Universität Hamburg  
Vogt-Kölln-Straße 30  
D-22527 Hamburg  
Germany

Apple is a registered trademark of Apple Computer, Inc.  
Java is a registered trademark of Oracle Corporation.  
JavaCC is a trademark of Oracle Corporation.  
L<sup>A</sup>T<sub>E</sub>X is a trademark of Addison-Wesley Publishing Company.  
LibreOffice is a trademark of The Document Foundation.  
MacOS is a trademark of Apple Computer Inc.  
Microsoft Office is a registered trademark of Microsoft Corporation.  
MySQL is a trademark of Oracle Corporation.  
Oracle is a registered trademark of Oracle Corporation.  
PostScript is a registered trademark of Adobe Systems Inc.  
Sun is a registered trademark of Oracle Corporation.  
T<sub>E</sub>X is a trademark of the American Mathematical Society.  
UML is a trademark of the Object Management Group.  
Unicode is a registered trademark of Unicode, Inc.  
UNIX is a registered trademark of AT&T.  
Windows is a registered trademark of Microsoft Corporation.  
X Windows System is a trademark of X Consortium, Inc.

The trademarks may be claimed in one or more countries.  
Other trademarks are trademarks of their respective owners.  
The use of such trademarks does not indicate that they can be freely used.

Please refer to the license section of the Renew user guide for more information about copyright and liability issues.

This document was prepared using the L<sup>A</sup>T<sub>E</sub>X typesetting system.  
This document is contained in the file doc/architecture.pdf as distributed together with Renew 5.0.



We are sorry, but we were not able to update the architecture guide to the major changes that came with Renew 2.0 and further architectural changes in recent versions. This guide still presents the architecture of release 1.6. However, the basic ideas and algorithms of the application are still the same, although some Java package names and interfaces have changed.

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Customizing Renew . . . . .	8
1.2	Before you Start . . . . .	9
1.3	Overview . . . . .	9
1.4	Acknowledgements . . . . .	9
<b>2</b>	<b>Package Overview</b>	<b>11</b>
2.1	Hierarchy <code>CH.ifa.draw</code> . . . . .	11
2.2	Hierarchy <code>de.renew</code> . . . . .	11
2.2.1	Simulation Engine . . . . .	13
2.2.2	Net Formalisms . . . . .	15
2.2.3	Graphical User Interface . . . . .	15
2.3	Package <code>de.uni_hamburg.fs</code> . . . . .	15
<b>3</b>	<b>Algorithms and Data Structures</b>	<b>16</b>
3.1	Anatomy of the Simulation Engine . . . . .	16
3.2	Unification . . . . .	17
3.2.1	Motivation . . . . .	17
3.2.2	Unknowns . . . . .	18
3.2.3	Backtracking . . . . .	18
3.2.4	Unifiability and Java Equality . . . . .	19
3.2.5	Occurrence check . . . . .	20
3.2.6	Calculations . . . . .	20
3.2.7	Tuple Index . . . . .	22
3.3	Semantic Level . . . . .	22
3.3.1	Net Structure . . . . .	22
3.3.2	Transition Inscriptions . . . . .	23
3.3.3	Expressions . . . . .	24
3.3.4	Some Expressions . . . . .	26
3.3.5	Some Functions . . . . .	27
3.4	Dynamic Level . . . . .	28
3.5	Event Handling . . . . .	30
3.6	Activated Transition Instances . . . . .	31
3.6.1	The Search Queue . . . . .	31
3.6.2	Triggers . . . . .	32
3.7	Search Algorithm . . . . .	33
3.8	Application to Petri Nets . . . . .	35
3.9	Finders . . . . .	38
3.10	Enacting a Binding . . . . .	38
3.11	The Shadow Layer . . . . .	41
3.11.1	Shadow Nets . . . . .	41
3.11.2	Net formalisms . . . . .	44

3.12	Simulation . . . . .	44
<b>4</b>	<b>How to Extend Renew</b>	<b>48</b>
4.1	Adding a New Arc Type . . . . .	48
4.2	Adding New Transition Inscriptions . . . . .	48
4.3	Adding a New Inscription Language . . . . .	49
4.4	Adding Graphical Figures and Tools . . . . .	50
4.5	Adding Simulation Statistics . . . . .	50
4.6	Adding Import and Export Filters . . . . .	50
<b>5</b>	<b>Java Bugs</b>	<b>51</b>
5.1	Graphics object loses draw commands . . . . .	51
5.2	Packing a frame is not portable . . . . .	51
5.3	Window titles are not shown correctly . . . . .	51
5.4	Memory leak through event objects . . . . .	51
5.5	Memory leak through windows . . . . .	51
5.6	Windows move on the screen . . . . .	51
5.7	Bad fonts and symbols . . . . .	52
<b>A</b>	<b>Glossary</b>	<b>54</b>

# List of Figures

2.1	Overview of all Renew packages . . . . .	12
3.1	Overview of the simulation core packages . . . . .	16
3.2	Unifiable objects . . . . .	19
3.3	A tuple index . . . . .	21
3.4	Static net data . . . . .	23
3.5	Expressions . . . . .	24
3.6	Functions . . . . .	27
3.7	Simulation state . . . . .	28
3.8	Place event handling . . . . .	29
3.9	Transition event handling . . . . .	30
3.10	The search queue . . . . .	31
3.11	Triggers and triggerables . . . . .	32
3.12	The searcher/binder/finder data structure . . . . .	35
3.13	A search process . . . . .	36
3.14	The <b>Finder</b> classes . . . . .	38
3.15	The class <b>Binding</b> . . . . .	39
3.16	The <b>Executable</b> classes . . . . .	40
3.17	The life cycle of an early executable object . . . . .	42
3.18	The shadow arc types . . . . .	42
3.19	The shadow classes . . . . .	43
3.20	The standard shadow compiler . . . . .	44
3.21	The simulator implementations . . . . .	45
3.22	The concurrent simulation algorithm as a Petri net . . . . .	47
4.1	Handling of textual inscriptions . . . . .	49

# List of Tables

3.1	Inscription classes and associated occurrence classes . . . . .	37
3.2	Occurrence classes and associated executable classes . . . . .	41
3.3	The simulator status codes . . . . .	45

# Chapter 1

## Introduction

The intention of this document is to give the interested user of Renew some rudimentary insight into the general structure and some individual components of Renew. This document does not aim to be a programming guide, but it tries to help you in finding a point where to start.

Note that, in general, you do not need to read this document. You can work with Renew perfectly without knowing its internals. Whenever you have a problem that does not seem to be solvable with the current version of Renew, think once more. If you still consider an extension of Renew essential, you should start with this manual before digging into the source.

To gain something from reading this manual you need considerable Java experience, the Renew source package, and willingness to read the source. Although large parts of the source are not yet documented, you might be able to infer most of the functionality from the names given to classes and methods, especially after checking this manual.

### 1.1 Customizing Renew

Because Renew is available with source, it is tempting to create a customized version of it. This is indeed possible and this manual will help you with the task, but there are a few drawbacks:

- Currently Renew is only partially commented.
- You diverge from the main development line. The internal structure of Renew might change significantly in future versions. If you send your additions/improvements to us, we will try to include them in the main release.
- This internal programming guide still needs considerable work to cover all aspects of Renew.

However, there might be reasons to modify Renew due to your particular applications.

If you implement a new net formalism, we suggest that you start a new Java package for it, e.g. `de.renew.evenbetter`. You might want to place the package in your own package hierarchy.

Not all customization can be done by simply adding more classes. Sometimes you might require some changes to the standard classes. It is a good idea to check such issues with the Renew team, because sometimes we might know best where to place this hook or that abstraction.

Although we do not require this as part of our license, we will be very happy if you send us any modified source code.



## 1.2 Before you Start

Before you start reading this manual, you should run JavaDoc to create the online documentation from the source files. This will help you in getting a feel of the system and in following this manual more easily. Many classes in the `de.renew` hierarchy do not contain JavaDoc comments, but you will still find it valuable to browse through the classes easily.

Since Renew uses the Java library, you should have learned everything there is to learn about the packages `java.lang` and `java.util`. You will not be able to proceed without that knowledge.

Since Renew uses Doug Lea's collection classes [4], you should familiarize yourself with this class library before looking at the simulation engine. The original distribution has some API documentation included.

Since Renew uses the Reflection API to execute Java inscriptions, you should familiarize yourself with the package `java.lang.reflect` before looking at the simulation engine. See the original documentation from Sun for details.

Since Renew uses the Java AWT for its windowing code, you should consult at least a tutorial about the package `java.awt`.

Since Renew uses the JHotDraw graph editing framework [1] for painting its nets, you should inform yourself about this package before investigating the GUI code. The classes in the `CH.ifa.draw` hierarchy contain nice JavaDoc comments and are a good place to start.

If you want to customize the inscription grammar, you should download JavaCC, the Java Compiler Compiler, from <http://javacc.dev.java.net/> which is free. We consider doing a complete rewrite of the grammar files in order to free them of the license restrictions that currently exist. In that course of work, it might be sensible to move from JavaCC (a great program) to ANTLR (a great program that is also free and available with source).

Class diagrams in this document are given in UML style notation. You are encouraged to learn about this notation from one of the many books or directly at [5]. In all diagrams we aim at providing the best overview and structural knowledge. To this end, the diagrams do not always include information about all attributes, but rather those attributes that are useful for understanding that part of the architecture that is currently discussed.

Sometimes we use attributes where one might rather expect an association or vice versa, but this is always done in order to simplify the diagram and to convey the intended meaning of a construction. For similar reasons, private attributes that are publicly accessible via getter and setter methods are sometimes shown as public attributes.

## 1.3 Overview

We will now explain the basic structure of this document. In Chapter 2 we give an overview of the packages that structure the Renew application.

## 1.4 Acknowledgements

We would like to thank Prof. Dr. Rüdiger Valk and Dr. Daniel Moldt from the University of Hamburg for interesting discussions, help, and encouraging comments.

We would also like to thank David Altenkamp, Nam Anh Duong, Sophie Bartelt, Levi Benecke, Beril Boran, Simon Bott, Sinan Brendel, Marvin Brendel, Bastian Butzke, Laif-Oke Clasen, Clara Dorothea von Barga, Leon Eisenblätter, Melissa Eisenburger, Sebastian Fumanek, Magdalena Genova, Lorenz Grimm Tim Groth, Arthur Hagen, Robin Huß Ayala, Robin Kirsch, Sam Knoop, Julian Köster, Lukas Krellenberg, Lasse Kuhnt, Stefan Le, Patrick Leonhardt, Eric Liebrecht, Maja Mercedes Perz, Justus Middendorf, Gavril Milchev, Marvin Müller, Michael Mutin, Paul Nedelmann, Mika Neumann, Jan Niklas Ritter, Fati Oboyse Tina, Kilian Patrick Maier, Shawn Ray Söker, Mica Ruben Langkat, Julius Ruchhöft, Philipp Schult, Fabian Slack, Elden Sokoli, Lennard Solterbeck, Yannik Stahl, Finn Stolten, Robin

Teichmann, Jan Vincent Hölzle, Eva-Lotte Vischer, Leven Wichelmann and Leon Zander Maxim Amrein for their work during the preparation of this release.

We would like to thank Jörn Schumacher for the prototype of the plug-in system (2.0), Benjamin Schleinzner for his work during the preparation of former releases (2.1-2.2) and Berndt Müller who has been of great help with respect to previous Renew releases for MacOS ( $\leq 2.0$ ). Some nice extensions of Renew were suggested or programmed by Michael Köhler-Bußmeier and Heiko Rölke.

We are indebted to the authors of various freeware libraries, namely Mark Donszelmann, Erich Gamma, Doug Lea, David Megginson, Bill McKeeman and Sriram Sankar.

Dr. Maryam Purvis, Dr. Da Deng, and Selena Lemalu from the Department of Information Science (<http://infosci.otago.ac.nz/>), University of Otago, Dunedin, New Zealand, kindly aided us in the translation of parts of the documentation and are involved in an interesting application project.

Valuable contributions and suggestions were made by students and scientific workers at the University of Hamburg, most notably Hannes Ahrens, Tobias Betz, Jan Bolte, Lars Braubach, Timo Carl, Dominic Dibbern, Friedrich Delgado Friedrichs, Matthias Ernst, Matthias Feldmann, Max Friedrich, Daniel Friehe, Olaf Großler, Julia Hagemeister, Sven Heitsch, Marcin Hewelt, Jan Hicken, Thomas Jacob, Andreas Kanzlers, Lutz Kirsten, Till Kothe, Annette Laue, Matthias Liedtke, Marcel Martens, Klaus Mitreiter, Konstantin Möllers, Eva Müller, Jens Norgall, Sven Offermann, Felix Ortmann, Martin Pfeiffer, Alexander Pokahr, Tobias Rathjen, Dennis Reher, Christian Röder, Heiko Rölke, Benjamin Schleinzner, Jan Schlüter, Marc Schönberg, Jörn Schumacher, Michael Simon, Fabian Sobanski, Volker Tell, Benjamin Teuber, Thomas Wagner, Matthias Wester-Ebbinghaus, Martin Wincierz, and Eberhard Wolff.

We also thank the following students for their contributions to the releases of Renew since version 4.0 Youssef Al Shriteh, Deert Bessey, Sven Billigmann, Marvin Brendel, Laif-Oke Clasen, Valerij Dobler, Kjell Ehlers, Marcel Hansson, Max Hartenstein, Alexander Heinze, Benjamin Hosseini, Karl Ihlenfeldt, Jan Robert Janneck, Jonte Johnsen, Svenja Junghans, Abdulssatar Khateb, Laszlo Korte, Tim Kowalczyk, Valentin Kroen, Jannik Kronziel, Clemens Kurz, Hamed Mohammadi, Patrick Mohr, Rana Mostafa, Justus Müller, Justin Pietsch, Morten Purwin, Tim Radke, Jannis Schuhardt, Malte Schuldt, Sven Schwartz, Alexander Senger, Elden Sokoli, Relana Streckenbach, Miriam Strulik, Marvin Taube, Kiro Vasilovski, Thorwin Vogt, Lukas Voß, Kevin Wessel, Christian Willner and Sven Willrodt.

We would like to thank the numerous users of Renew who provided hints and constructive criticism. They helped greatly in improving the quality of the code and the documentation. In particular, we would like to name Alun Champion and Zacharias Tsiatsoulis.

## Chapter 2

# Package Overview

This chapter introduces the main packages and their use. In Fig. 2.1 you can see the dependencies among the various packages. Here, as always in the packages diagrams in this manual, dependencies that can be inferred via a sequence of other dependencies are usually not displayed, even if a direct dependency exists, too.

### 2.1 Hierarchy `CH.ifa.draw`

Although the class hierarchy presented here still reflects the general structure of the original JHotDraw [1], it has been heavily modified. Especially, a parent/child mechanism was added to associate figures with each other. Moreover, a single graphical editor may now be responsible for multiple drawing windows.

**`CH.ifa.draw.framework`** This package contains the main abstractions of JHotDraw in the form of interfaces and some auxiliary classes.

**`CH.ifa.draw.standard`** This package contains some basic implementations of the abstractions found in **`CH.ifa.draw.framework`**. Typically, these implementations require some additional code to be fully functional, but they ensure that tedious bookkeeping tasks are taken care of and that some convenience methods are available.

**`CH.ifa.draw.figures`** This package contains classes for figures of various shapes, but also the tools and handles required to create and modify these figures.

**`CH.ifa.draw.contrib`** This package contains code for a number of figures that were not originally part of JHotDraw.

**`CH.ifa.draw.util`** This package contains miscellaneous classes. Most of these are concerned with building the GUI of JHotDraw and doing I/O. The class **`Storable`** is especially notable, because it supports the external storage format for saving drawings.

**`CH.ifa.draw.application`** This package contains a complete graphical editor as a stand-alone application. The main class is **`DrawApplication`**.

In many classes of this hierarchy, events are used for propagating state changes. Make sure to understand, at least vaguely, the event mechanism and all of the interfaces in the package **`CH.ifa.draw.framework`** before digging deeper into the sources.

### 2.2 Hierarchy `de.renew`

This hierarchy constitutes the Petri-net-specific part of Renew. Two subhierarchies can be found here: **`de.renew.formalism`** and **`de.renew.gui`**, which are responsible for implementing

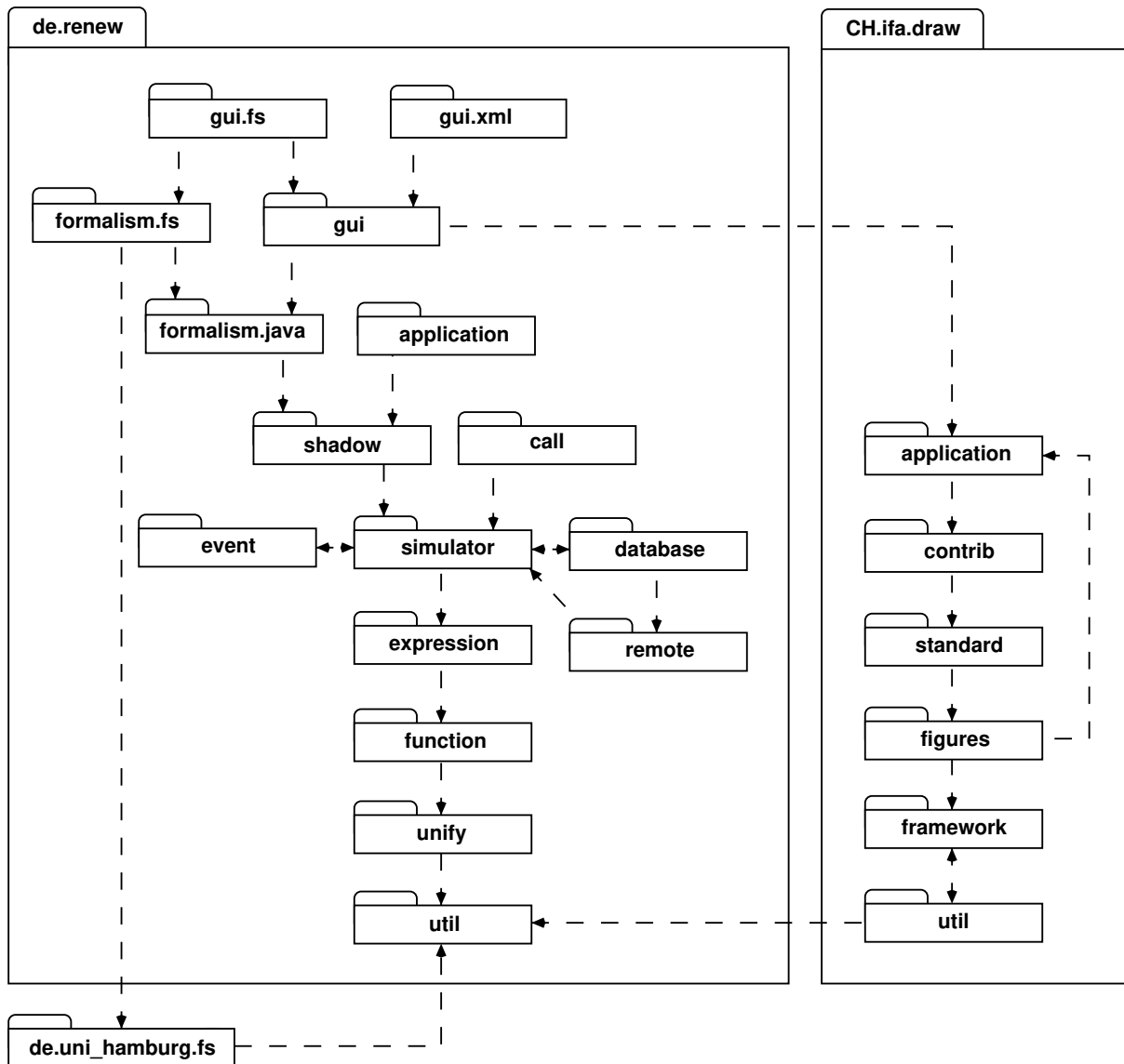


Figure 2.1: Overview of all Renew packages

specific net formalisms and for implementing a GUI, respectively. But first we discuss the other packages in this hierarchy, which constitute the core simulation engine.

### 2.2.1 Simulation Engine

These packages are concerned with the non-graphical representation and execution of high-level Petri nets.

**de.renew.application** This package contains applications that allow the non-graphical stand-alone simulation of Petri nets. At the moment, **ShadowSimulator**, which bases the simulation on a serialized shadow net system, is the only class here. Other applications might follow.

**de.renew.call** This package supports nets that implement Java methods. This is done by automatically generating subclasses of **de.renew.simulator.NetInstance**, so-called Net Stubs, that convert method calls to synchronous channel invocations. These invocations are subsequently stored in the usual search queue and executed by the simulator as soon as possible. The Renew User Guide contains a description of how to utilize Net Stubs.

**de.renew.event** This package contains the event mechanism that is used to couple the simulator and the GUI. It provides interfaces for event listeners and producers and an abstract convenience implementation for listeners.

**de.renew.function** This package contains a variety of function objects. An implementation of the **Function** interface must provide a single method that converts its argument into a result. There are predefined function objects to perform all of the transformations of objects that can be specified in a Java expression. This package is not yet concerned with variables or assignment, but only with operations on runtime objects and values.

**de.renew.shadow** The shadow layer separates the GUI from the execution layer. Shadow nets represent the net drawings, but they abstract from all information that does not influence the simulation, like position, size, or color of the net elements.

This package is based on a create/discard API: You can create net elements, but few changes can be applied to these elements without discarding and recreating them. In fact, even discarding shadow nets is discouraged. Instead, an application should create a new shadow net each time this is required.

The implementations of **ShadowCompiler** are responsible for creating nets at the semantic layer using the shadow net.

**de.renew.remote** This package allows to access the state of a running simulation remotely via RMI calls. In some sense, it will be the counterpart of the shadow package. While the **de.renew.shadow** package allows it to convert net drawings to nets, this package will facilitate the display of net instances in net instance drawings.

Renew  
**1.6** This package is now fully functional. The gui packages do no longer access simulator classes directly to display a running simulation. Instead, all state information is obtained through remote accessor objects.

**de.renew.unify** This package encapsulates a unification algorithm. Unification is done in-place using modifiable objects. The unification algorithm supports a backtracking mechanism using the class **StateRecorder**. Ordinary Java objects are unifiable if and only if they are equal according to **Object.equals(...)**. Variables can send notifications after they become fully bound, so that functions can be evaluated as soon as their arguments are fully known. An occurrence check is performed. The special class **Calculator** of

unifiable objects is unifiable only with itself, but is incorporated into the occurrence check. These objects are used to represent calculations that can be based on some objects and might lead to an arbitrary result.

**de.renew.util** A package of miscellaneous classes that are used somewhere else, but were considered too useful across applications.

**de.renew.expression** This package combines the functionality of the **de.renew.function** package and the **de.renew.unify** package and implements full Java expressions. It adds those expressions that cannot be interpreted by a mere function evaluation. Especially, this package handles the introduction of local variables. To do this, every expression is evaluated in the context of a variable mapper, which maps variable names to variables of the unification mechanism. Other additions of this package are type checks, constants, and bidirectional expressions.

**de.renew.engine.common** This packages defines some reusable occurrences and executables. These objects are not sufficient for defining a formalism, but they constitute essential elements for most formalisms.

**de.renew.engine.searcher** This package contains the basic algorithms that are used during the search for an activated binding. Classes and interfaces remain very abstract here: We talk about things that may be searched and things that may be executed, but do not give any concrete examples.

**de.renew.engine.searchqueue** This package defines the **SearchQueue**, which is responsible for keeping track of possibly enabled **Searchables** so that they can later on be searched for activated binding. The search queue is also keeping track of the current simulation time, so that it may order the searchables according to the earliest possible time when a search may possibly succeed.

**de.renew.engine.simulator** A simulator makes sure to retrieve searchables from the search queue and to search them for activated binding. the binding are then executed with a policy suitable for the simulator, be it concurrently or sequentially.

**de.renew.net** This package defines the basic building block for defining nets and net instances: places and transitions. It does not yet deal with arcs and transition instances.

**de.renew.net.arc** Here we define various arc types for use with nets.

**de.renew.net.event** Events and event listeners are used when dynamic changes of a net's state and of actions in the net must be tracked by other code.

**de.renew.net.inscription** Transition inscriptions augment a transition's behavior. The inscriptions defined in this package are by no way exhaustive, but represent the most commonly used inscriptions.

**de.renew.database** This package interacts with the simulator core and makes sure that all changes in all nets can be recorded in a database. The access to the database is transactional, i.e., the database driver is notified about all tokens that are moved by a single transition in a single method call and is supposed to record these changes permanently and atomically.

**de.renew.watch** This package allows to keep track of all possible valuations of a synchronous channel of a net instance. Afterward, one valuation may be chosen to request an explicit firing of a transition. This has been applied to design a workflow engine, but other uses are imaginable, as it allow an external entity to watch and control the flow of the simulation.

### 2.2.2 Net Formalisms

`de.renew.formalism.java` This package provides a compiler for the Renew default net formalism that inputs shadow nets and outputs nets at the semantic layer. Most other net formalism are based on this package. Typically, only the parser needs to be exchanged for a different one in order to accommodate for a different syntax.

`de.renew.formalism.stub` A variant of the Java net formalism that experiments with some improved type rules, which are essential when generating net stubs. Usage of this net formalism is discouraged.

`de.renew.formalism.bool` A special net formalism that implement Boolean Petri nets. These nets are especially useful to experiment with workflows that are derived from event-driven process chains.

`de.renew.formalism.fs` This formalism was a first prototype to include Feature Structures (see below) into Renew. It is now only used as a basis for the following formalism and for debugging.

`de.renew.formalism.fsnet` This formalism supports Feature Structures as a means to describe object constraints. It includes a type modelling tool to easily specify used defined types. Also, all available Java classes can be used as types. Tokens are Feature Structures which can be combined using unification and can be tested for subsumption.

### 2.2.3 Graphical User Interface

`de.renew.gui` This contains the GUI for drawing, editing, and simulating Petri nets with Renew. Specialized figures aid in the presentation of the net and new tools modify these figures. The application class `CPNApplication` is responsible for setup and coordination.

`de.renew.gui.xml` This package contains an experimental XML storage format for nets. The XML import function uses the visitor pattern and requires a SAX-compatible XML parser.

`de.renew.gui.maria` This package features a special mode for drawing nets of the Petri net analyzer Maria. No simulation support is available, but it is possible to design a net graphically and export it to simulation in Maria.

`de.renew.gui.fs` This package provides the GUI elements needed in the FSNet formalism (see above). These are type modelling figures and Feature Structures. The Feature Structure figure is also used to render Java objects as **expanded Tokens**.

## 2.3 Package `de.uni_hamburg.fs`

This package is an implementation of a typed Feature Structure formalism and has been developed for Renew's `FSMode`. A Feature Structure is a graph with typed nodes and labelled arcs. The arc labels can be seen as features or attributes of the nodes. A Feature Structure points to a root node of such a graph. The type system allows subtypes (*is-a*-relations) and incompatible types (*is-not-a*-relations). Types can be tested for equality and subsumption. Among the functions available for Feature Structures is subsumption and unification, which is in fact graph unification. The packages `de.renew.formalism.fs`, `de.renew.formalism.fsnet`, and `de.renew.gui.fs` depend on this package.

## Chapter 3

# Algorithms and Data Structures

In this chapter we will give an introduction to the most important algorithms of the simulation engine. Fig. 3.1 summarizes those packages mentioned in Fig. 2.1 that are highlighted subsequently.

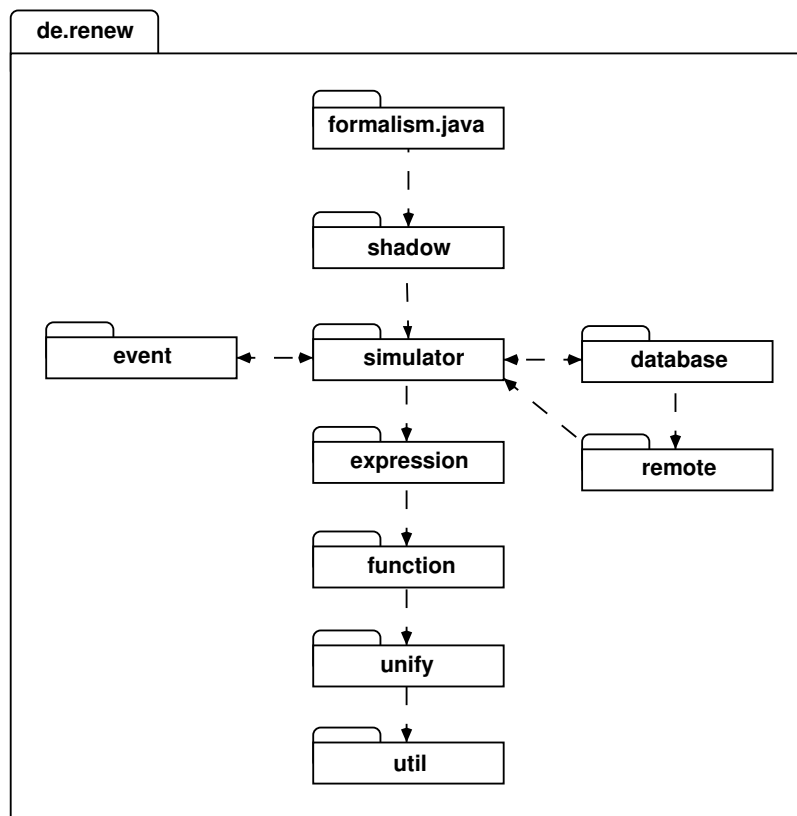


Figure 3.1: Overview of the simulation core packages

### 3.1 Anatomy of the Simulation Engine

The implementation of a Petri net simulator can be separated into various aspects, which can be regarded independently from each other to some extent:



- A unification algorithm. A large part of the complexity of a Petri net simulator can be alleviated through the consistent use of a unification algorithm. Because there are special requirements, a dedicated algorithm has to be written.
- The data structures for the representation of the net structure and the net state. This is a particularly meaningful point, because each simplification, in addition, each special treatment on this level immediately impedes the other parts of the algorithm. Since we do not want to produce a code individually for a net, the data structure must be particularly flexibly adaptable, in order to be able to represent all [sorts of] possible Petri nets. Since net instances should be able to be simulated, also the description of a current marking is more complex than for ordinary Petri nets.
- The selection of a transition for check on activation. If a transition was not detected as activated, it should only be checked again, if a realistic chance exists that it could be activated.
- The search for enabled bindings of an individual transition. This is the central part of the algorithm, which should be able to find bindings as many as possible [and] as fast as possible. This algorithm can become very complex in concept, in particular if there are many optimization considerations, but it should however be simple, as there it has a special influence on the robustness of the whole system.
- The firing of a transition. Basically the execution of a transition is comparatively a simple process, as soon as an enabled binding is found. If we allow however the parallel execution of several transitions, conflicts develop with the access to common data here. Here the powerful synchronisation methods of Petri nets must be illustrated on concepts of a programming language.

In the following sections we will examine each item individually.

## 3.2 Unification

First of all we describe the data structures and algorithms of the unification algorithm. All classes used for this algorithm can be found in the package `de.renew.unify`.

### 3.2.1 Motivation

For improved usability, the inscription language of a Petri net formalism should support some sort of tuples to facilitate the easy retrieval of matching values. Tuples are already a classic area in which unification algorithms have been applied, and the unification of tokens at places with arc inscriptions requires at least a matching algorithm. There are, however, aspects of the simulation algorithm for which a unification algorithm is useful in different ways.

Petri net formalisms often require the consideration of many different transition inscriptions when firing a transition. Mostly, these inscriptions are given without a particular order, so their effect should not depend on the order of their evaluation. This coincides with the characteristic of unification algorithms, i.e., the sequence of unification is irrelevant.

For synchronous channels, no direction of the information flow is prescribed. during the search for enabled bindings, the values and variables on both sides must be unified, as a simple assignment is generally not possible. First of all, a parameter of a synchronous channel itself could be again a tuple expression, if we want to keep the orthogonality of the language definitions. Further, a synchronous channel might have to be handled even before the last variable of the initiating side is bound. This is especially important when the channel transfers values in both directions.

The question is, whether more unifiable objects, besides tuples, should be examined. Lists were identified as a reasonable extension of a net formalism. Although lists can be represented

as nested tuples, for the sake of better usability this way of implementation should not be externally visible. However, we cannot modify the representation of tuples, as nested tuples do not always represent a list. Hence a special category of lists was created which allows a more suitable representation and prohibits the visibility of their internal structure as well. In the following, we will not deal with this class often since nothing changes in the actual unification algorithm and all interesting effects can already be observed with tuples.

### 3.2.2 Unknowns

Each variable has a value. This can be a normal Java object or a unifiable object. When a new variable is generated, its value is unknown at first, because the variable is completely unbound. In order to be able to indicate a value nonetheless, special objects have been introduced by the class **Unknown**.

These objects become important during unification. After two unassigned variables **x** and **y** have been unified, their value is still unknown. The unification, however, is visible in that both variables would return the same unknown as their value.

We note that, after the unification of **x** and **y**, it is not specified which of the two unknowns will form the later value of **x** and **y**. In fact, this is irrelevant for further unifications. Outside the unification algorithm, the class **Unknown** need not be known anyway, because variables should be queried for their valuation only after they have become completely bound, i.e., when no unknown is part of their value, not even nested within a tuple.

### 3.2.3 Backtracking

For the implementation of a unification algorithm one can choose one of two ways. Either unification creates a new binding list that assigns the appropriate values to the variables, whereby the original bindings list is preserved; or, alternatively, the old binding information is overwritten and thus no longer available.

Here, the second way, which modifies the existing objects, has been chosen, so that unnecessary copying can be avoided.

In a Petri net simulator, however, it is necessary to be able to reset to the old state as required, for example when a proposed binding of an arc variable to a token value did not lead to an activated transition and alternative bindings should be tried out.

Therefore all modifications that the unification algorithm executes are noted in a central object, which belongs to the **StateRecorder** class. For all modifying accesses to unifiable objects a recorder must then be specified. If the special value **null** is used, then the corresponding operation cannot be undone. Otherwise, all attributes of the object that are supposed to be modified are stored in the recorder. It is important to make no modification at all before recording it first.

In order to keep the recording of the modifications as flexible as possible, the state recorder does not prescribe a format for the information that must be recorded. Instead, an object of the type **StateRestorer** is transmitted. This object possesses only one method **restore()** and stores all necessary information. For each type of modification a subclass of **StateRestorer** is created.

Several reset points can be specified for a recorder, so that a partial resetting is possible. Modifications are always undone in the reverse order in which they were made.

It will be shown that almost the entire state of the Petri net simulator is stored in unifiable objects, and that almost the entire backtracking can be performed by this algorithm. If information is not connected to unifiable objects, special subclasses of **StateRestorer** can handle these cases, too.

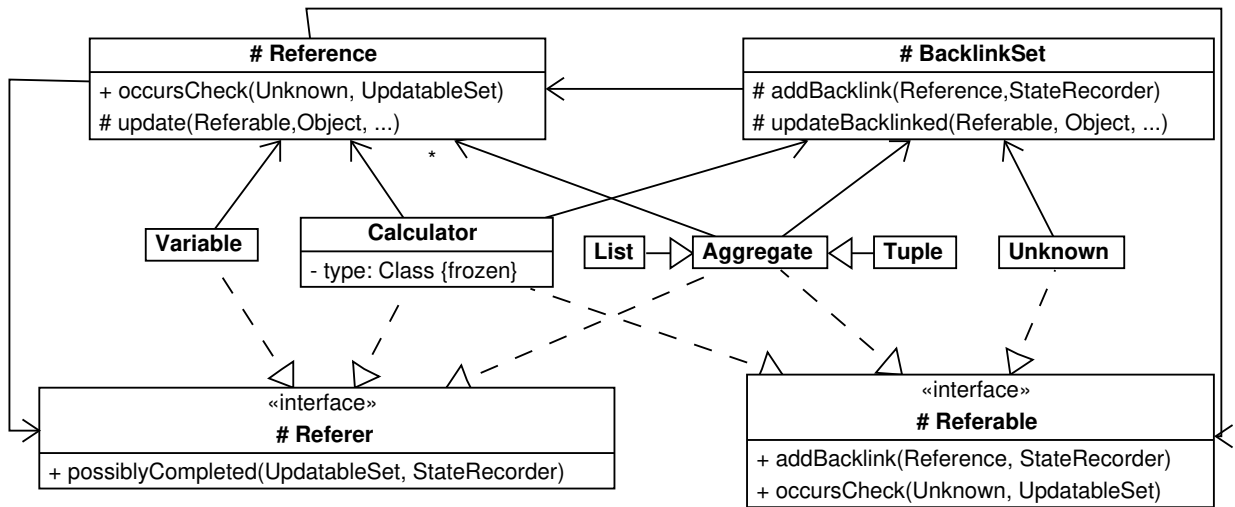


Figure 3.2: Unifiable objects

### 3.2.4 Unifiability and Java Equality

The unifiability of Java objects was implemented on the basis of the method `equals(Object)`. Thus we follow the decision of the programmers of container classes, where this way is also always chosen to access the contents of a container. It is a procedure already embodied in the language definition, which can be adapted sufficiently flexibly to individual needs.

The substantial problems develop from the fact that the Java definition of equality allows `equals(Object)` to vary over time. In most cases the Java API follows the rule that only for unchangeable objects equality may be coarser than identity, but there are exceptions as for instance `java.awt.Point` or the container classes from Java 1.2.

This is to be borne, if Java's equality concept is normally used. It becomes a serious problem for unification, however, because a check for equality may occur in many unexpected places.

Now the question arises, how unifiable objects themselves deal with testing on equality. Tuples are very easy to handle, because for them the equality is defined by the equality of all components.

For unknowns, however, a certain problem results, because they can still take any value by unification, so that the outcome of a comparison with other objects is not at all defined.

Therefore a comparison attempt on unknowns throws an exception, which interrupts the normal program flow and is normally announced as error. Indeed a comparison should not occur, because unknown should be used only under the control of the Petri net simulator. To other program sections the simulator should only pass completely bound tuples, so that unknown should not be accessible from normal Java code.

Since backtracking operations can again introduce unknown components into already fully unified tuples, it would be possible that an appropriate method stores reference to a complete tuple and later, after the backtracking, accesses the tuple again and retrieves an unknown. Since those methods that are not invoked from within actions should not have any side effects, this effect will only occur in actions. In that case all, however, all tuples will first be copied, so that they are not subject to backtracking any more.

A hash code must be assigned to each Java Object. For tuples this is calculated using a simple polynomial derived from the hash codes of the tuple components. For unknowns the query of the hash code throws an exception, because an unknown should not be stored in a hashed data structure.

### 3.2.5 Occurrence check

If a tuple could contain itself directly or indirectly, then one could describe certain infinite data structures quite easily. But such things are not easy to handle in the mathematical theory and cause problems during the implementation, too.

With unification further problems occur. In particular, attention would have to be paid to avoid endless loops. Also unification is founded on the basis of term unification of predicate logic, where infinite terms are not allowed.

Thus there is the task is to ensure cycle-freeness during the unification: the so called occurrence check, which checks for the occurrence of a tuple within in another tuple. The occurrence check is sometimes not implemented in the area of logical programming, and one leaves the behavior in the case of recursive tuple unspecified, because a check in that context would be very expensive. In Petri net simulators the principal complexity is due to other algorithms, so that an occurrence check does not slow down the simulator considerably.

### 3.2.6 Calculations

In the formalism of the reference nets it is possible to carry out certain calculations only during the execution of a transition, which is noted with the keyword `action`. The results of the calculations are not known to the unification algorithm, yet the algorithm should as far possibly be able to deal with these calculations.

Especially, late calculations should be effectively executable during the transition's firing. This leads to certain requirements:

- Cyclic dependencies shall be detected. This applies also to complex multi-level dependencies. For example, the call `action x=[1,a.method(x)]` should fail before the firing of the transition, because here `x` depends on itself indirectly through a method invocation and a tuple.
- As far as possible, the result type of a later calculation should be represented by unification algorithm, so that no preventable type errors may occur.

A calculation is represented by special unifiable objects of the type `Calculator`. A calculator is unifiable only with unknowns and with itself, but not with tuples, values or other calculators. In particular, equality reduces to identity for calculations.

Calculator objects reference exactly another object, which can serve as an argument for a calculation. If more arguments are required, this can be implemented by a calculator object that references a tuple object.

Occasionally a variable value must be of a certain type, in order to be a valid allocation for the variable. this is ensured by the class `TypeConstrainer`. Such an object monitors an arbitrary value. As soon as the value is no more an unknown, the type of the new value is checked. This might be possible before the value is completely bound. For example, a tuple may be type checked before all its components are bound.

In order to be able to provide type checking for late calculation, all calculator objects carry the predicted type of their result. If a `TypeConstrainer` detects as a calculation object as value, the predicted type is used instead of the type `Calculator`.

In Fig. 3.2 we summarize the main classes involved in the representation of unifiable data structures. You can see how every implementation of `Referer` is assisted by an instance of `Reference`. Similarly, every `Referable` is augmented by a `BacklinkSet`. A backlink set collects information about all those references that reference its owner. A reference makes sure to insert itself into the backlink set of its referenced object.

Using a `CalculationChecker` object a program can require that certain variables must be bound or that they must be complete. A value is complete if it contains no unknowns, even nested within a multitude of unifiable objects. A complete value is bound if it contains no calculators.

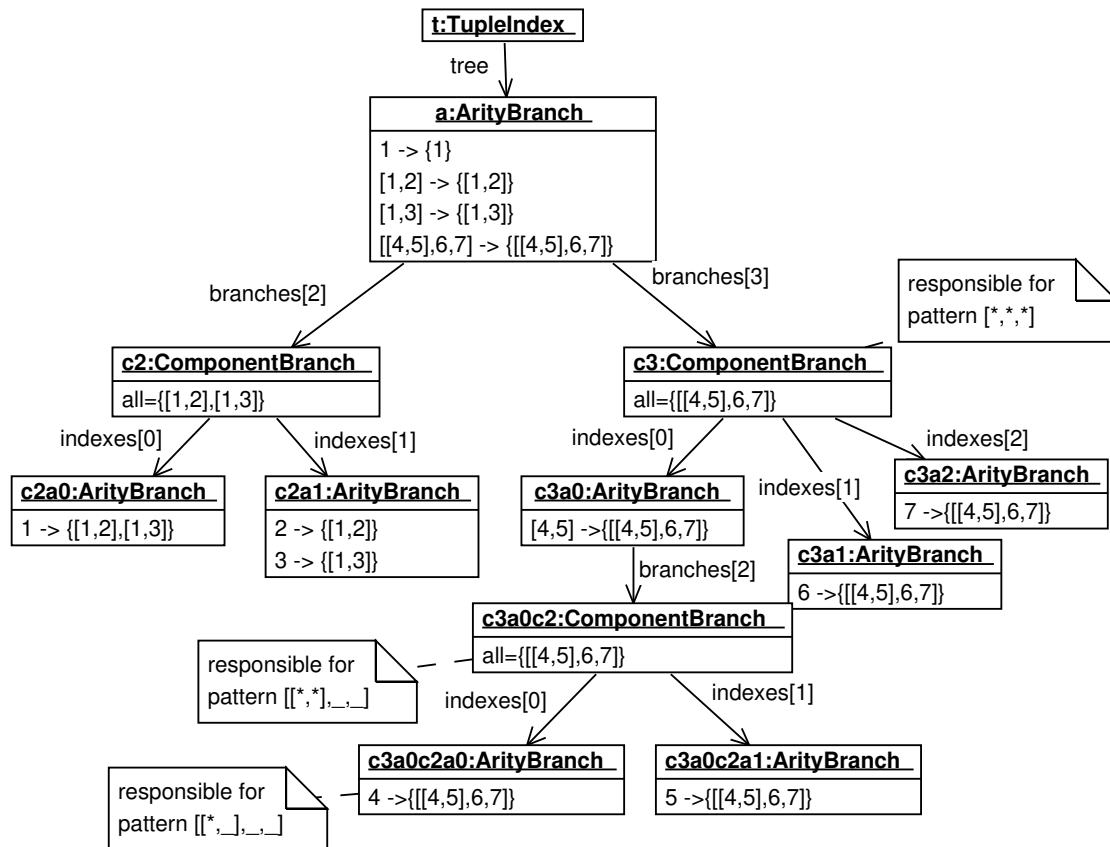


Figure 3.3: A tuple index

### 3.2.7 Tuple Index

The tuple index is a specialized data structure that allows to select among a set of tuples some candidates that might fit a given pattern. The given pattern is itself a possibly nested tuple, which might be incomplete, i.e., it might contain unknown in some substructures.

The tuple index provides an upper bound for the set of matching values based on exactly one component of the tuple or one component of a subcomponent of the tuple. The tuple index will try all complete subcomponents of the tuple during lookup and select the best estimate among these. It will not, however, consider more than one components. E.g., with the set  $[0,0,0]$ ,  $[0,1,0]$ ,  $[0,1,1]$ , and  $[1,0,0]$  of values and the pattern  $[0,0,\_]$ , the only matching pattern is  $[0,0,0]$ , but the optimal guess based on the second component contains  $[0,0,0]$  and  $[1,0,0]$ .

## 3.3 Semantic Level

Now we investigate the data structures of the semantic level where textual annotations have already been resolved, but no instances have yet been built.

### 3.3.1 Net Structure

Whenever possible, we prefer a component-oriented architecture in the following sense: The functionality of an object results substantially from aggregated sub-objects and attribute values, where the allocation of the sub-objects and the values occurs at run-time, but it remains constant for the life span of the aggregate. The opposite of this would be an architecture, in which the functionality of a class is adapted by the creation of sub classes. This would not be so flexible, because it would make independent variations of different aspects more difficult, as Java does not allow multiple inheritance.

Since no code is generated for a net, all semantic information about a net has to be represented in a data structure. The application domain naturally suggests the classes **Place**, **Transition**, and **Net**. A net must know about all its net elements, so that places and transitions can be taken into account during the generation of a net instance.

Places possess an initial marking. This is represented by an arbitrary number of objects of the type **TokenSource**. Each of these objects produces a token for the place during initialization. In the simplest case, a token source could simply return a constant during a call to the method `createToken()` as done by the class **ConstantTokenSource**. Alternatively, it might be required to evaluate an expression, as in the class **ExpressionTokenSource**.

Now arcs and transition inscriptions must be represented. We observe that these two groups of objects are quite similar, because they influence the enabledness and the effect of a transition. Therefore, we want to abstract from the graphical difference of arcs and inscriptions at this point, so that both can be treated as special transition inscriptions. The class **TransitionInscription** is the common superclass of all transition inscriptions. A transition can aggregate as many objects of this class as necessary.

Thus a certain asymmetry in the handling of places and transitions develops, since arcs are mainly associated with transitions. This view is to be found however not so rare at all. Even with S/T nets arcs are occasionally represented by pre-and post- region functions for transitions. Further there are the transitions, which must consider at one time all their arcs, while this is not the case with places, so that due asymmetry already exists in the formalism.

In Fig. 3.4 you can see a class diagram for the semantic layer of the net representation. Most of the transition inscriptions are listed, where three of them require special handling: **UplinkInscription**, **CreationInscription** and **Arc**. In the implementation, the class **Arc** is divided into a number of specialized classes that takes care of the different arc types.

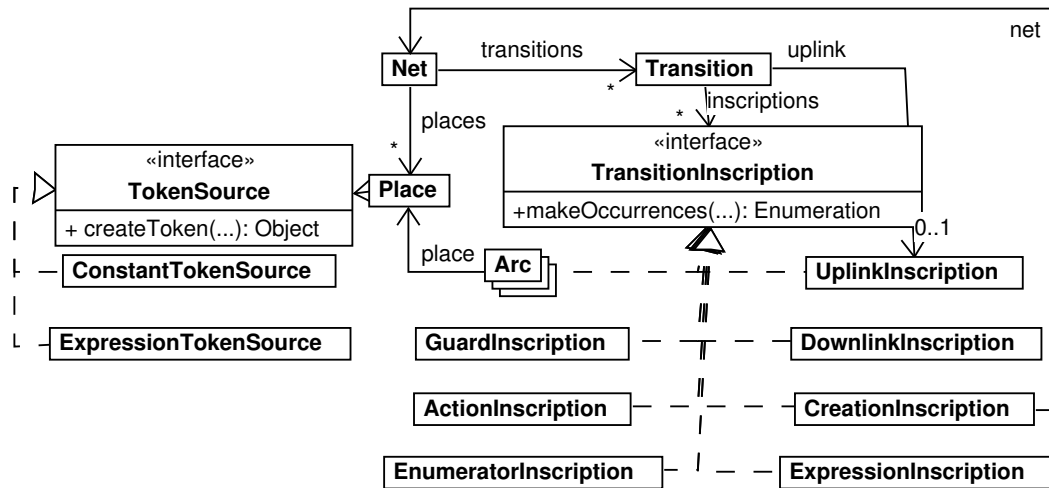


Figure 3.4: Static net data

### 3.3.2 Transition Inscriptions

In the following, we give an overview of all transition inscriptions that were shown in Fig. 3.4. All transition inscriptions implement the interface **TransitionInscription**.

**Uplinks** An **UplinkInscription** determines on which channel incoming synchronisation requests must appear. Since this information is referred to frequently, each transition stores a reference to its unique uplink explicitly, if such a reference exists. Each uplink has a name and an expression, which is evaluated during a synchronisation in order to determine the channel data.

**Downlinks** Additionally to the attributes of an uplink, a **DownlinkInscription** possesses another expression, which is evaluated to the object that has to provide the uplink for the synchronisation.

**Net creation** **CreationInscription**-objects reference the net that is supposed to be instantiated. It would also be possible to store only the name of the net, but that would introduce the possibility to interchange the implementation of a net dynamically, which might be dangerous.

**Expressions** An **ExpressionInscription** object encapsulates an expression, which should be evaluated during the search for an enabled binding. The result of the evaluation is discarded.

**Guards** **GuardInscription** objects behave like expressions, but they force the evaluation result to be **true**.

**Actions** **ActionInscription** objects behave like expressions, but they are evaluated only while the transition fires.

**Arcs** All arc types reference a place and an arc expression, and pay attention to whether the movement of tokens is to be logged. Additionally, the class **Arc** can distinguish different types of arcs, so that we do not need to implement different classes for input, output, test, and reserve arcs. On top of that, the class **ClearArc** manages a type for the array that is created during the processing of the arc. The class **FlexibleArc** adds knowledge about two conversion functions, with which the values supplied at the arc are converted into concrete tokens.

A couple of times we mentioned expressions in this section. Expressions allow to parameterize inscriptions.

### 3.3.3 Expressions

Expressions can occur in two contexts:

- in action inscriptions, where no evaluation is to take place immediately, but a registration of the pending calculation is required, and
- in other transition inscriptions like guards, where the result is required as soon as possible.

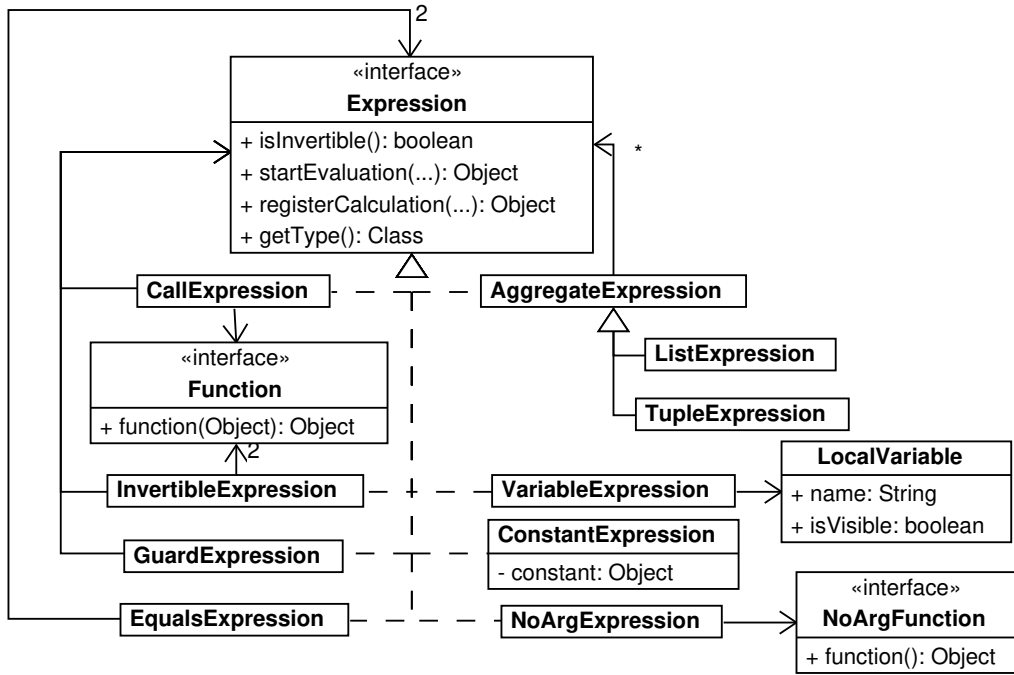


Figure 3.5: Expressions

Even in the first case the expression has to be evaluated during the firing of the transition in exactly the same manner as it would be necessary for the second case. Hence we suggest to treat both types of evaluation in a single class. With the use an expression in a general transition inscription a part of that functionality lies idle, but a consistent handling is guaranteed in both cases.

For the internal representation of expressions the interface **Expression** is defined. Together with some implementations this class belongs to the package `de.renew.expression`. The most important methods are the registration of an evaluation during an action and the actual evaluation.

Both of the two methods `registerCalculation` and `startEvaluation` need three arguments:

- a `VariableMapper`-object, which maps the name of a variable onto the associated variable of the unification algorithm,
- a `StateRecorder` object, which can undo all modifications executed during the evaluation of this expression,



- a **CalculationChecker** object, with which late calculations and requirements for the early availability of a result can be announced.

The different implementations do not always use all arguments for their evaluation. For example the evaluation of a variable or a constant does not need a **StateRecorder**, because only read accesses occur on unifiable objects.

On the other hand the **VariableMapper**-object is necessary only for the evaluation of a variable. Its responsibility is to return for a given **LocalVariable** object a variable of the unification algorithm. It has to ensure that the mapping from **LocalVariable** objects to variable is consistent for different branches of an expression.

An expression can be used with different **VariableMapper**-objects each time it is evaluated. The algorithm ensures that no variable is reused from earlier evaluations.

However, most expressions are associated to subexpressions that must be evaluated first, before they themselves can be evaluated. Such expressions aggregate other expressions and pass on their arguments to the sub-expression with the call of the evaluation methods.

Each evaluated expression returns the result object immediately, even if it is a unifiable object that is not yet fully evaluated. An evaluation might even return an **Unknown**-object to signal a totally unbound variable. Therefore all assembled expressions must expect that their arguments, which were returned by the subexpressions, are not yet ready and that the computation of the expression must be deferred. Therefore the concrete calculation is encapsulated in an object of the type **Notifiable**, which is registered with the unification algorithm. As soon as all arguments are fully bound, the object is notified and can evaluate the expression and unify the result with its result variable. If all subexpressions result in definite values, the notification is sent immediately.

Caution is necessary if more than one sub-expression is analysed. The return value of the first sub-expression could be an **Unknown**. This **Unknown** could be unified with a value during the evaluation of the second expression. Although this is a quite unusual case, which arises almost only with artifacts like  $[x, x=1]$ , it has to be considered.

We store therefore first the **Unknown** in a variable, because variables automatically take care of unknown values that acquire a value through unification. A normal Java reference could not achieve this. This is the price which we must pay for the usage of the unification algorithm.

Since both Java references and primitive objects can result from the evaluation, a suitable result type has to be chosen. The decision was to encapsulate primitive Java values in objects of the class **Value**. The solution used in the Java reflection API (`java.lang.reflect` package) to convert primitive types into their object counterparts without wrapping does not preserve the distinction between the different types. Although one of the advantages of primitive types, namely their run time efficiency, is no more given, this approach enables the type-correct handling of primitive values. Since **Value** is a reference type, **Object** can generally serve as return type for expressions.

Apart from the calculation methods two query methods are to be implemented. An **Expression**-object can specify a result type, in order to permit a type check. It is guaranteed that every result of a successful expression evaluation belongs to this result type.

After the evaluation of an expression the result may be undetermined, because the evaluation is deferred at least partly due to unbound variables. The method `isInvertible()` indicates whether variables occurring in the expression can be bound by unifying the incomplete result with a fixed value. This applies in particular when the expression only consists of one variable at all, but also in the case of tuples that are built from subexpressions.

This concept of invertibility is naturally related to the mathematical invertibility, but here it refers to the concrete operational feasibility. Additionally, we will not inversely calculate some expressions, if thereby the clarity of the formalism would suffer.

This query on invertibility is used by input arcsin particular, since the possible result values in form of the tokens in the place are already certain in this case. Here it is sensible

to try all possible values one after the other, but only if this could lead to variable bindings based on the structure of the expression at the input arc.

### 3.3.4 Some Expressions

Not all expressions are to be discussed in detail, since they are to a large extent straightforwardly coded, but some classes are exceptional.

An **EqualsExpression** possesses two sub-expressions, which are to be unified during the actual calculation. It would be possible to unify them already during the registration of a subsequent calculation, like in **action** `x=y`. The only question is whether this would be the intended semantics. Because in **action**-inscriptions we would like to achieve a value transfer that is close to the evaluation rule of Java, we prescribe a value transfer from the right to the left. This is achieved by unifying the left side with a calculation object that references the result of the right side's registration, where the latter result is possibly unknown during registration.

**TupleExpression**-objects aggregate a sequence of expression-objects, and can generate and return tuples. In a very similar way, lists are generated by **ListExpression**-objects, so that a common super class, **AggregateExpression** can be found.

As indicated, **VariableExpression**-objects draw their results from the **VariableMapper**-object released during the evaluation. Noteworthy is it here, that variables produce the only communication possibility between different inscriptions.

Method invocations are administered by **CallExpression**-objects. Method invocations often have many different arguments and are sometimes static and sometimes dynamic. It would thus appeal to have a flexible approach that delays an evaluation until a multitude of expressions has been completely evaluated.

The situation becomes simpler, if **CallExpression**-objects aggregate only a single object of type **TupleExpression**, which summarizes all arguments into one, so that only one argument, which arises out of one subexpression, must be processed by the **CallExpression**-objects.

During the evaluation this sub-expression is evaluated and the result is stored in a variable. Likewise for the result of the **CallExpression**-object a variable is produced, which contains an **Unknown** initially.

If by means of **registerCalculation()** only the registering of a late calculation is required on the basis of an action inscription, a **Calculator**-object of the unification algorithm is generated.

On the other hand, the method **startEvaluation()** forces the calculation of the function as soon as its argument tuple becomes known. To this end, an observer object is registered at the argument variable. The observer object will be notified as soon as the result of the sub-expressions is completely determined. It calculates the function and unifies it with the result variable.

If the result variable is already bound at the point of time when the function is evaluated, it will be checked by the unification algorithm, whether the newly calculated result corresponds with this value.

In order to abstract from different ways of calling a method or a constructor, the core functionality is shifted into objects of the type **Function**. During its evaluation a function receives exactly one object and returns one object. The **CallExpression**-object can be limited to the supporting activities: the construction of the argument and the registration for notification.

The contents of the result variable are finally returned to the higher expression and can be further used there, no matter whether the function could be evaluated already or not.

One version of the **CallExpression** is the **NoArgExpression**. This needs no argument values and aggregates no sub-expressions either. This is typically used to read a static attribute of a class. Such an expression aggregates no ordinary function, but a **NoArgFunction**-object.

Casts are implemented invertible, as far as it is possible. To achieve this, the class `InvertibleExpression` can calculate at the same time two functions during the evaluation. The first function is handled as by a `CallExpression`-object. The second function will be calculated, as soon as a notification queues up, that the result of the expression is known and the result is unified with the argument value.

### 3.3.5 Some Functions

The presented `Function`-concept can even be expanded onto other calculations. Here we want to list some possible functions.

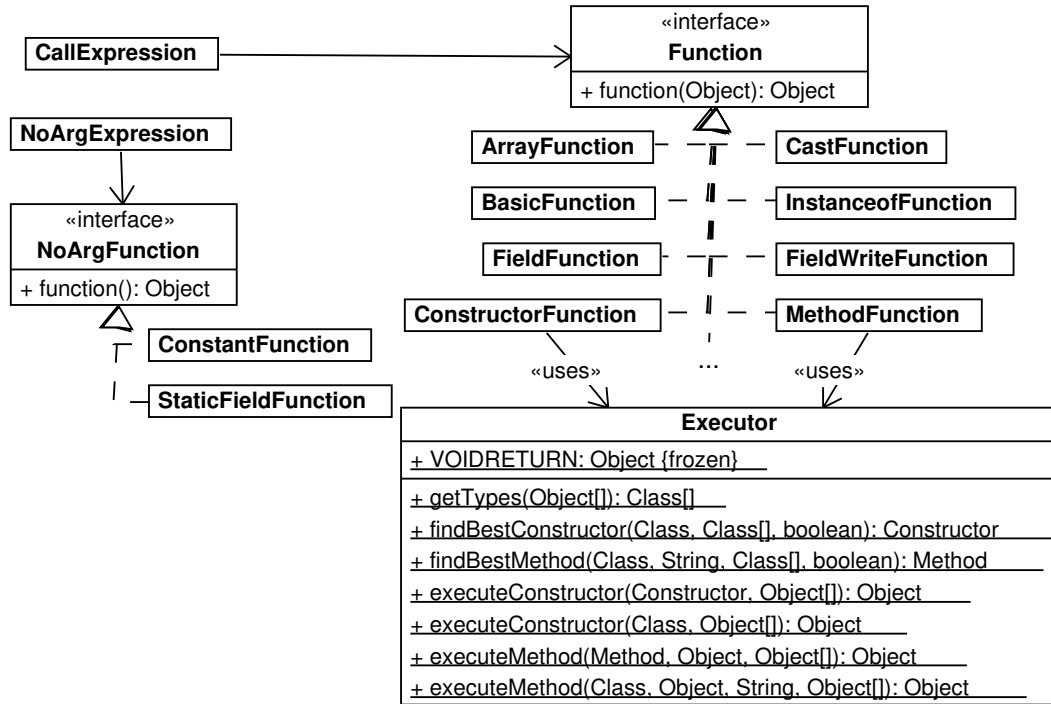


Figure 3.6: Functions

**Dynamic method invocations.** These require a pair of an object and an argument tuple.

**Static method invocations and constructor calls.** Here only the argument tuple is important.

**Reading and writing of attributes.** During the writing it is to be noted that a side effect evolves, so that these operations are only meaningful in action-inscriptions.

**Calculation of operators.** Also primitive operators such as addition or multiplication can be regarded as special functions. Here finally there are (only) many function objects, which were implemented as singletons.

**Casts.** Some of the previous operations could fail, if an exception is thrown by the calculation, this is however particularly obvious with casts. Functions can release therefore the exception `Impossible`. Usually this brings the caller to the stage, in which the last operation, which has bound a variable, is recognized as illegal and a backtracking sets in.

Other functions are conceivable and easy to set up. The advantage of this approach over the programming of a subclass of **Expression** is that for functions no knowledge about the unification algorithm or the simulation algorithm is necessary.

The Reflection API of Java is used for the execution of many functions. Hereby the method signatures of each desired class are inspected and method invocations are issued. In order to find an appropriate method for given parameter types, all methods must be checked whether their signature matches the argument types and the most specific method must be found from the appropriate methods. This happens at run-time in the untyped formalism or at compilations time for the typed formalism.

As mentioned earlier, primitive values are always encapsulated in **Value**-objects, both on the level of expressions and on the level of functions. This is to be taken into account during the implementation of the functions.

### 3.4 Dynamic Level

Apart from the static aspects of a net, the dynamic state of net instances must also be stored, so that all required information is available during the search for an enabled binding. Normal Petri nets only need the current marking, which could always be stored together with the static information about the places.

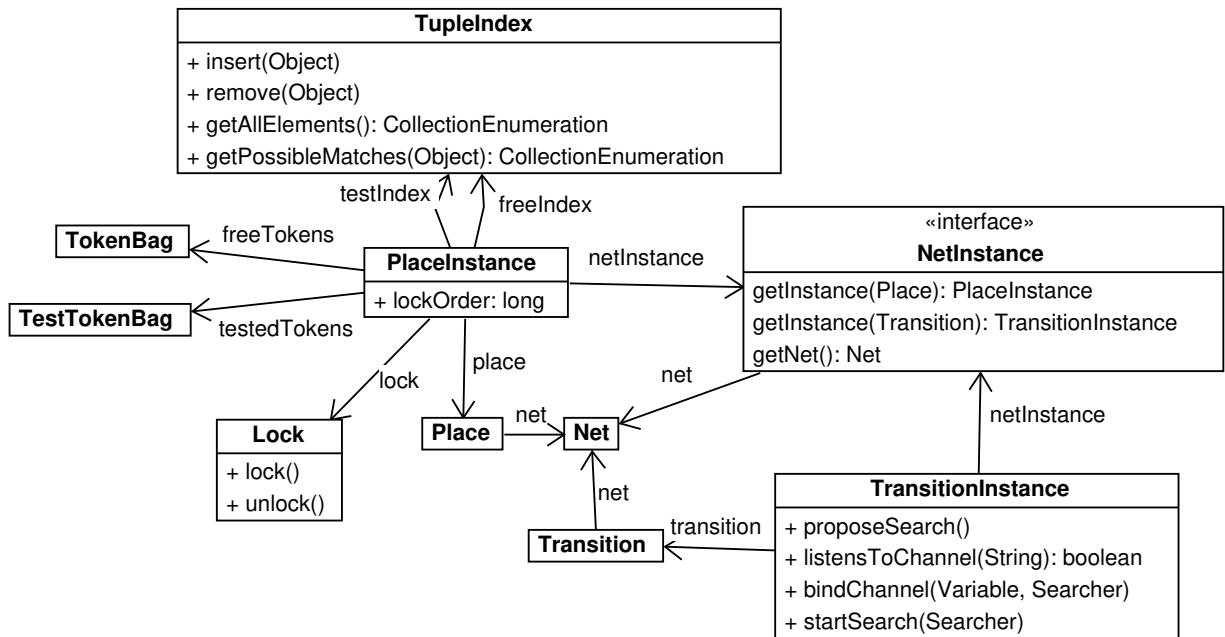


Figure 3.7: Simulation state

As for reference nets however, several net instances are to be built from each net, so that the markings of the different net instances must be managed separately from each other. For this an interface **NetInstance** with a standard implementation **NetInstanceImpl** is created, in order to handle the state of a net. In addition, a **NetInstance** object denotes the identity of a net instance. The net is associated to the net instance during the entire life-time of the net instance. It contains a unique ID for the net, so as to generate readable trace outputs.

It is one responsibility of a **NetInstance** object to enable places to produce place instances by means of the method **getInstance(Place)**. Each place instance contains a multi-set of free tokens and a multi-set of tested tokens, which are tested by means of a test arc. Since

the firing duration of transitions is not limited, we cannot expect that tested tokens become free again quickly. Rather they must be handled explicitly in the search algorithm for test arcs.

For each place instance there are two tuple indices: one for the free tokens and another for the testable tokens. To this end, all tokens that either lie free in the place or are already tested are regarded as testable. Both indices are implemented by the class `TupleIndex`.

The reason to indicate not only the tested tokens, is that then for test arcs both indices would have to be queried and the results would have to be combined. Instead we rather invest into the modification of two indices during the movement of free tokens. The place instance is responsible for the correct update of the indices during each modification of the marking.

Each place instance contains a lock mechanism, which protects it against parallel accesses. Before each access that concerns a variable attribute of the place the caller must make a call on `lock()`. It is necessary that the caller controls the lock, because it happens that multiple tokens must be moved, which is best achieved by locking first and then doing a sequence of updates.

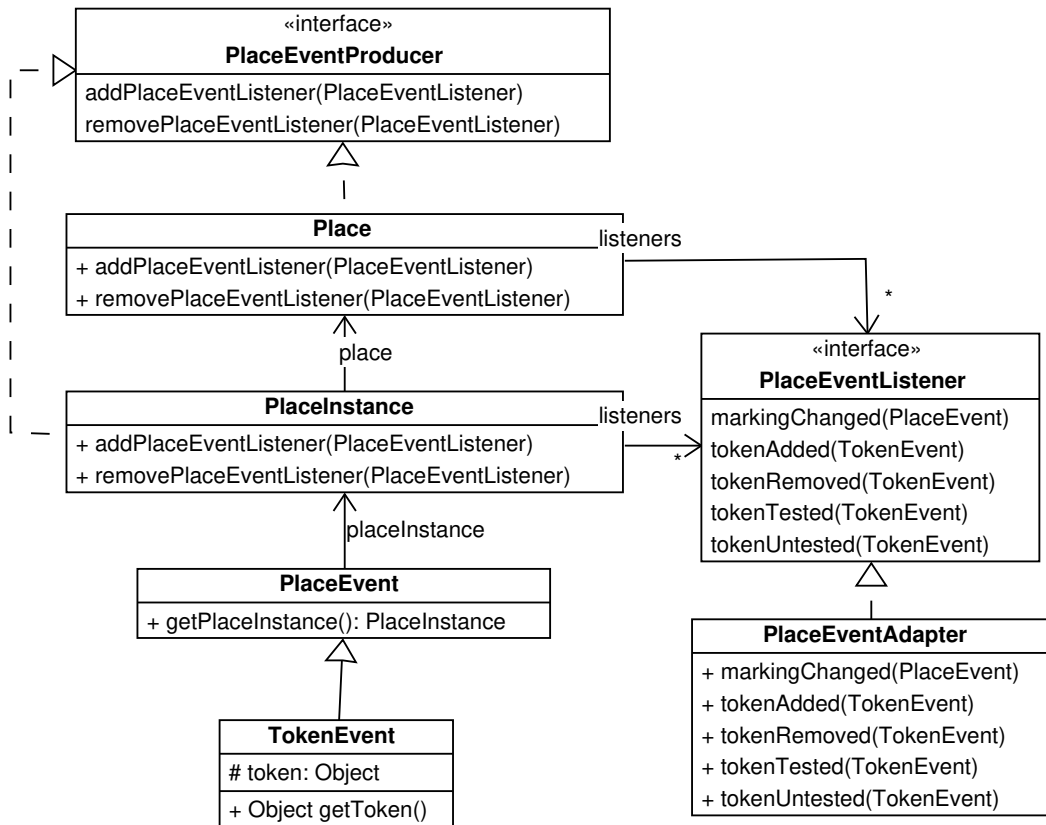


Figure 3.8: Place event handling

It is also possible that two threads operate jointly, but not concurrently on a place instance. This might happen if remote methods are invoked during an operation. Because both threads need access to the place and only one can acquire the lock, it must be allowed that a thread accesses the place instance without having gained the lock. That means the access methods must not lock themselves.

### 3.5 Event Handling

The event mechanism allows event listeners to monitor the current state of a place or transition. Whenever a token is moved into a place or out of a place instance or becomes tested or untested, the place instance sends events to all those listeners that were previously added to it.

Similarly a transition instance sends events at the start and at the end of every firing. Currently transition instances do not send event when their enabledness changes, because the computational cost associated to this operation would be prohibitive.

Not only place instances and transition instances, but also the places and transitions themselves accept listeners. These listeners are notified about all events within all instances associated to this net element.

Unless a listener is deregistered from an event producer, it will receive all future events. No listeners are automatically removed.

Events are delivered synchronously, i.e. the simulation blocks while an event is processed by the listener. That means that listeners should typically terminate quickly. During the notification, the place instance is locked by the notifying thread.

It is safe to query the current marking of the place instance that produced the current event, but it is not allowed to modify its marking. While querying, it is not required to relock the place instance.

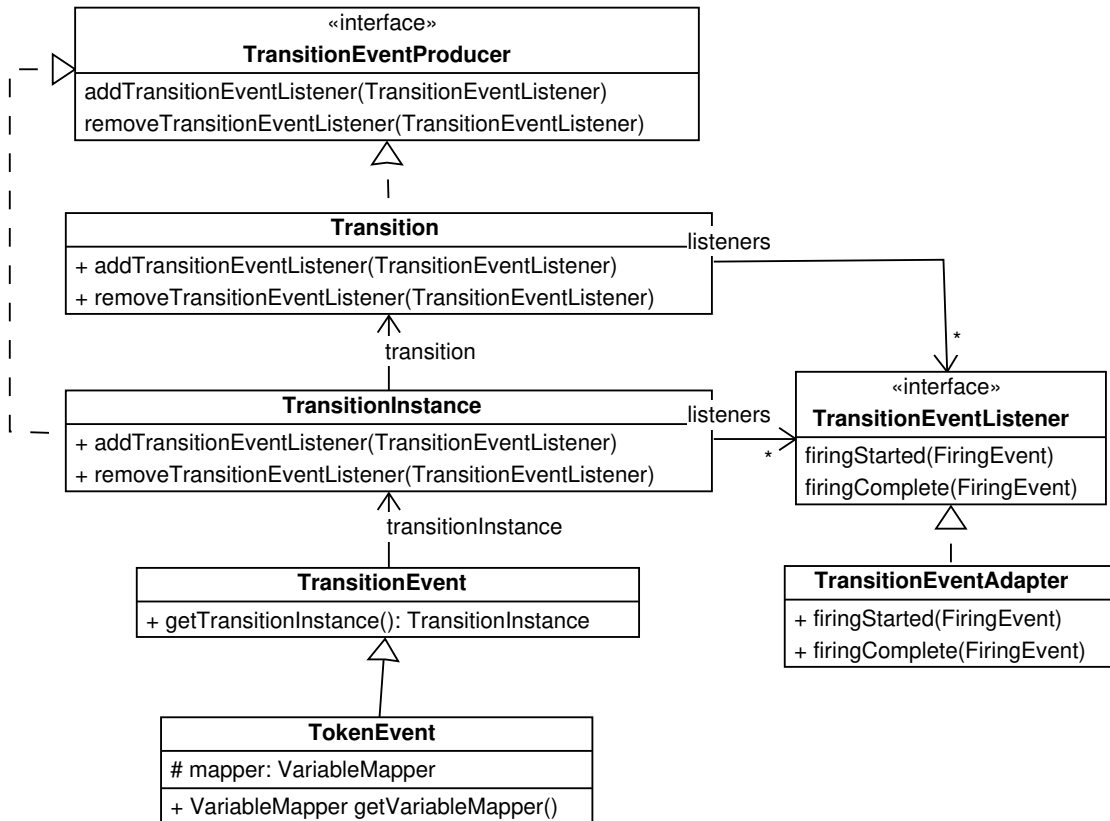


Figure 3.9: Transition event handling

Producers are the interfaces of classes that keep track of listeners. Listeners receive events. Events carry attributes that specify the precise kind of action that triggered the event. Adapters are standard implementations of listeners. Listeners receive events. See

Figs. 3.8 and 3.9 for detailed class diagrams.

## 3.6 Activated Transition Instances

Before computing the enabled bindings of a transition instance, the simulator need to determine which transition to search.

### 3.6.1 The Search Queue

It is often the case in an object-oriented Petri net that the enabled transition instances are located in a small sub-range of the net for longer time, whereas in other net parts no transition instances are activated. In such a case it would not be efficient to check all transition instances for enabledness every time. Rather those transitions are to be checked, whose enabledness status might have changed.

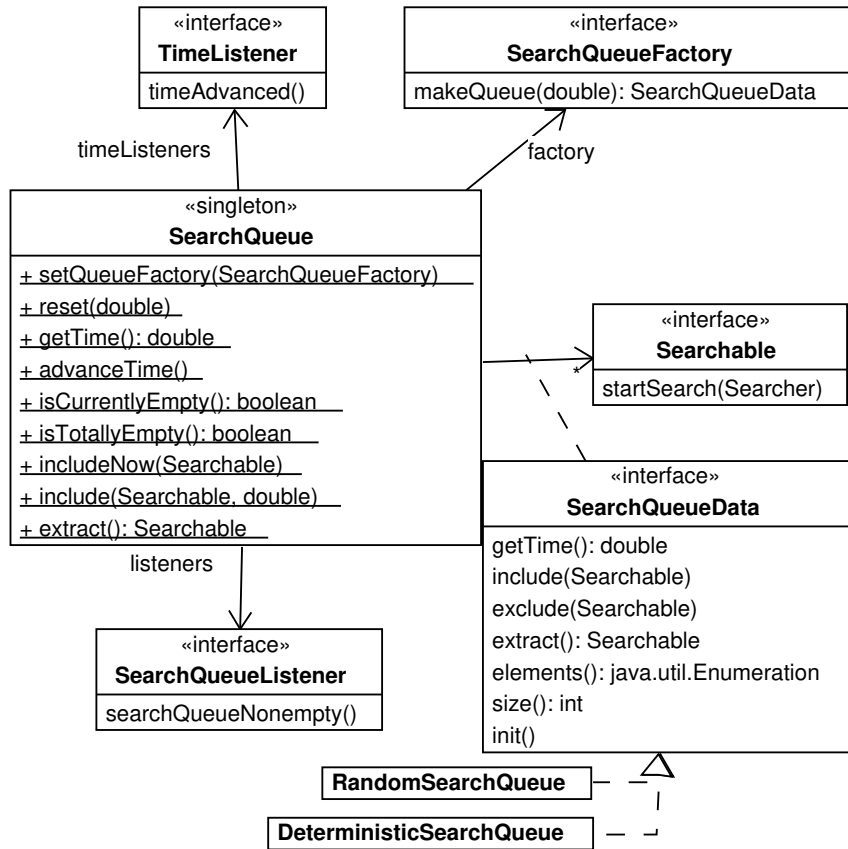


Figure 3.10: The search queue

To this end we keep all potentially activated transition instances in a central data structure. This data structure is implemented by the class **SearchQueue**. In a search queue, you can not only include transitions, but all kinds of objects that might have to be searched for activated bindings, indicated as implementations of the interface **Searchable**.

The search queue also keeps track of the time during the simulation of timed Petri nets. To this end, it records for each searchable object the earliest time when it should be searched. Whenever there are no more searchables that should be searched right now, the search queue

advances the time, notifies optional listeners and returns a searchable object for the next relevant point of time.

For each different time stamp, the search queue creates an object of type **SearchQueueData** which group the associated searchables. Different implementations of **SearchQueueData** can use different queueing strategies. The **SearchQueueFactory** is responsible for creating new instances of **SearchQueueData**-objects.

The simulator extracts possibly activated transition instances from the search queue and checks for enabledness. If the transition instance is not activated, the transition instance is discarded and another transition instance is selected. If it is enabled, it is reinserted into the search queue for a check on additional concurrent firings. If the simulator determines that the transition will be enabled at some future point of time, but not right now, it will be inserted with an appropriate time stamp.

The search queue may also notify listeners whenever new searchables arrive in the queue. For an overview of the search queue architecture, see Fig. 3.10.

### 3.6.2 Triggers

We observe that a transition can be recognized as deactivated without looking at all input places. For example input places could be empty or contain only such tokens that violate a guard. If the marking of a place that does not influence the enabledness changes, this event should no cause an insertion of the transition instance into the search queue, because the transition instance must still be disabled.

Only those places, whose tokens received attention during the check on activation at all, should provoke the new check. In order to realize this optimization, it is necessary that each place manages a set of the transitions, which have queried the place during their check on activation and which therefore must be notified after a modification.

The event mechanism discussed here is called the trigger mechanism. A place instance is a **Trigger** that might cause a transition instance to be rechecked. Hence a transition instance implements the interface **Triggerable**. Fig. 3.11 provides a class diagram for the trigger mechanism. The class **TriggerCollection** is a utility class that simplifies the administration of triggers.

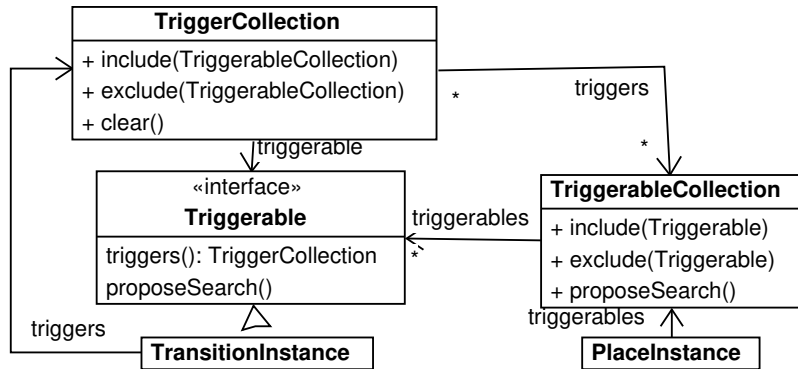


Figure 3.11: Triggers and triggerables

As soon as a notification is sent to all triggerables, the set of triggers can be deleted. All relevant transition instances have been inserted into the search queue and do not need another notification. Only while the transitions are again checked, they must log on themselves again for notification at the place instance. They will do this only if the place is relevant for the activatability of the transition instance.



Additionally, every transition keeps track of all its triggers. Having been triggered, it can explicitly deregister itself from all triggers, even those that did not cause the check for enabledness. As a modification was already detected, notifications by the other place instances are useless. Only during the new check for activation, the places whose current tokens are relevant should again be registered.

This way, the relevant place instances on which a transition instance depend may change over time. This is also helpful for good memory management, because links between triggers and triggerable might stop the garbage collector from claiming all unused memory otherwise.

We summarize the process:

- *Before* relying on a marking for the analysis of a transition enabledness, the transition is registered with the place's triggerable collection as a triggerable.
- In any case, a triggerable is registered at a triggerable collection, never vice versa.
- After a change is done to a place, the place's triggerable collection is used to inform all transitions of their possible enabledness.
- Each transition removes itself from all its triggers. Afterward, it inserts itself into the search queue.

In this process, the following lock order is respected to avoid deadlocks:

- First, lock triggerable collections via synchronisation.
- Second, lock trigger collections via synchronisation.

### 3.7 Search Algorithm

Here we describe the algorithm that determines enabled bindings for a transition. The basic algorithm is quite general and constitutes essentially an adaptable backtracking tree search.

During the tree search the decisions to be made at a certain point of time arise mainly out of input arcs that may be bound to different tokens. But there are other decision such as the choice of an appropriate transition that satisfies a given uplink.

Each of these decisions will typically lead to a variable binding, so that backtracking is necessary for examining the other possibilities. If many decisions are to be met, not all the possible combinations should be exhaustively searched, but rather the search should be aborted as early as possible.

All sorts of decisions are encapsulated in the interface **Binder**. An instance of the class **Searcher** coordinates the search process and selects the binder that makes the next decision. The searcher also references all information necessary for the search procedure. After all binders have had their chance to bind variables, the searcher determines whether the current set of variable bindings leads to an activated transition and transfers control to an instance of **Finder**, which may use the current bindings in the desired way. The main step before invoking the finder is to ask the **CalculationChecker** about possible conflicts with regard to the action inscriptions and the complete binding of all variables involved in early computations.

The search process is initiated by an object of type **Searchable**. Such objects own a method that can register some binders at the searcher and transfer control to the searcher.

After the Searcher is assigned to start a search procedure, it asks all binders for a estimate on the relative cost of trying all bindings in order to determine the binder that should start the search process. This cost is referred to as the binding badness. The searcher always selects the binder with the minimal cost.

A special value for the binding badness is reserved, in order to represent infinite costs. A binder should indicate infinite costs, if at this time no new binding information can be acquired from the binder. Otherwise it is a good clue, to indicate the number of necessary branches in the search tree as binding costs. Thus those binders are preferred by the searcher,

which keep the width of the search tree small. Since this does not ensure the optimal search tree in all cases, a binder can also use any other heuristics.

The selection of the optimal binder is highly dynamic. The order of binders may not only be different for different searches of one transition instance. It may in fact be different for different branches of the search tree. In order to exploit the performance advantages associated with this optimization, it is recommended that the computation of the binding badness itself should be relatively fast.

Typically binders with the binding costs 0 are those binders, which can prove that the current branch of the search tree does not contain enabled bindings. It is clear that such binders should be called with priority, so as to cut off the search tree as early as possible. Similarly, binding costs of 1 pertain to a binding, which must be executed in each case, since there are no alternatives. Such binders are likewise preferred, because their handling must be done in any case and can perhaps help to better estimate the costs of the remaining binders.

After the searcher determined a binder in such a way, it removes the binder from its list of unprocessed binders and transfers the control to the binder. Now the binder tries all possible bindings that may lead to a solution one variant after the other. In each case the resulting binding information

Each time, when the binder selected a possibility, the search method of the searcher is called again. The searcher can then determine and call one of the remaining binders. Thus binders and searcher alternate, until a solution is found, or until a binder ascertains that no decision can lead to success on its part. In such a case the binder terminates its search method and returns with an appropriate message to the calling searcher. This will go back to the last binder, which examines a further binding possibility.

A binder may not only contribute information to the search, but it may register further binders at the searcher for future consideration. Thus a decision could be made, but at the price of a new pending decision. These additional binders can make an initially simple search process complicated. But it also helps to keep a search process simple as long as no absolute necessity exists for a certain decision.

If a binder returns, because it did not succeed in finding an appropriate solution to the search problem, backtracking must occur. Modifications that were applied to variables of the unification algorithm can be easily cancelled, since the unification algorithm already introduced a backtracking mechanism, as described earlier. The **StateRecorder**-object, which is necessary for each modifying operation, is centrally administered by the searcher, since exactly one such object per search procedure is necessary.

Another modification during the traversal of the search tree concerns the registered binders themselves. Here the searcher makes sure that a binder is removed from the list of the possible binders before control is passed to it. After the search procedure it is automatically registered again. If a binder would like to register other new binders, then also the deregistration has to be administered by the binder.

Binders can rely on the following assumptions:

- Each binder is processed at some time, unless it reports an infinite binding badness.
- A binder is not called as long as it announces an infinite binding badness.

On the other hand the following duties fall upon binders:

- The sequence, in which two binders with finite binding costs are processed, may only affect the order, but not on the set of found solutions.
- By the actions of binders only restrictions, but not extensions of the remaining solution space, can take place.
- The binding costs, which a binder reports, may only decrease with the operations of other binders, but never increase.

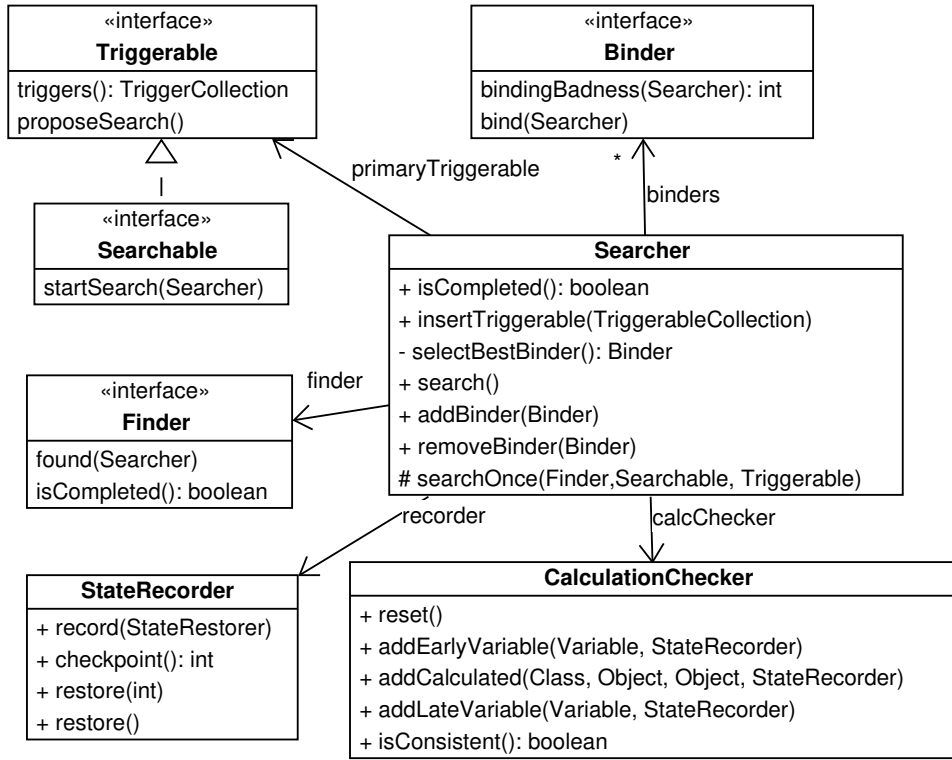


Figure 3.12: The searcher/binder/finder data structure

- Before a Binder accesses the status of a changeable object, in particular a place instance, it registers the currently searched transition instance as triggerable of the changeable object.
- A modification of a changeable object may lead to missing or redundant solutions, but not to other failures.

We assume in the following that binders communicate over the variables of the unification algorithm. Thus no central object is necessary, which must administer the binding information already collected. This enables us to combine binders flexibly.

Since variables are changed only by unification, they are only more strictly bound during the search process, so that the set of the allowed bindings decreases monotonously, as required by the specification of binders. Because the sequence of unification operations does not influence the outcome, it will also be simpler to achieve the arbitrary exchangeability of operations of binders.

The searcher accepts a solution, if all binders are processed. The searcher will then transfer the finder for evaluation, which can use the solution as desired. Finally the finder informs the searcher, whether further solutions are to be found. For this a query method `isCompleted()` is provided, which states, whether the search procedure is to be terminated.

In Fig. 3.13 we depict a typical search procedure as a collaboration diagram.

### 3.8 Application to Petri Nets

The search algorithm that was described up to now could be flexibly amended for many different application areas, since there is nothing intrinsically net-related about it. We will discuss the specialization to Petri nets now.

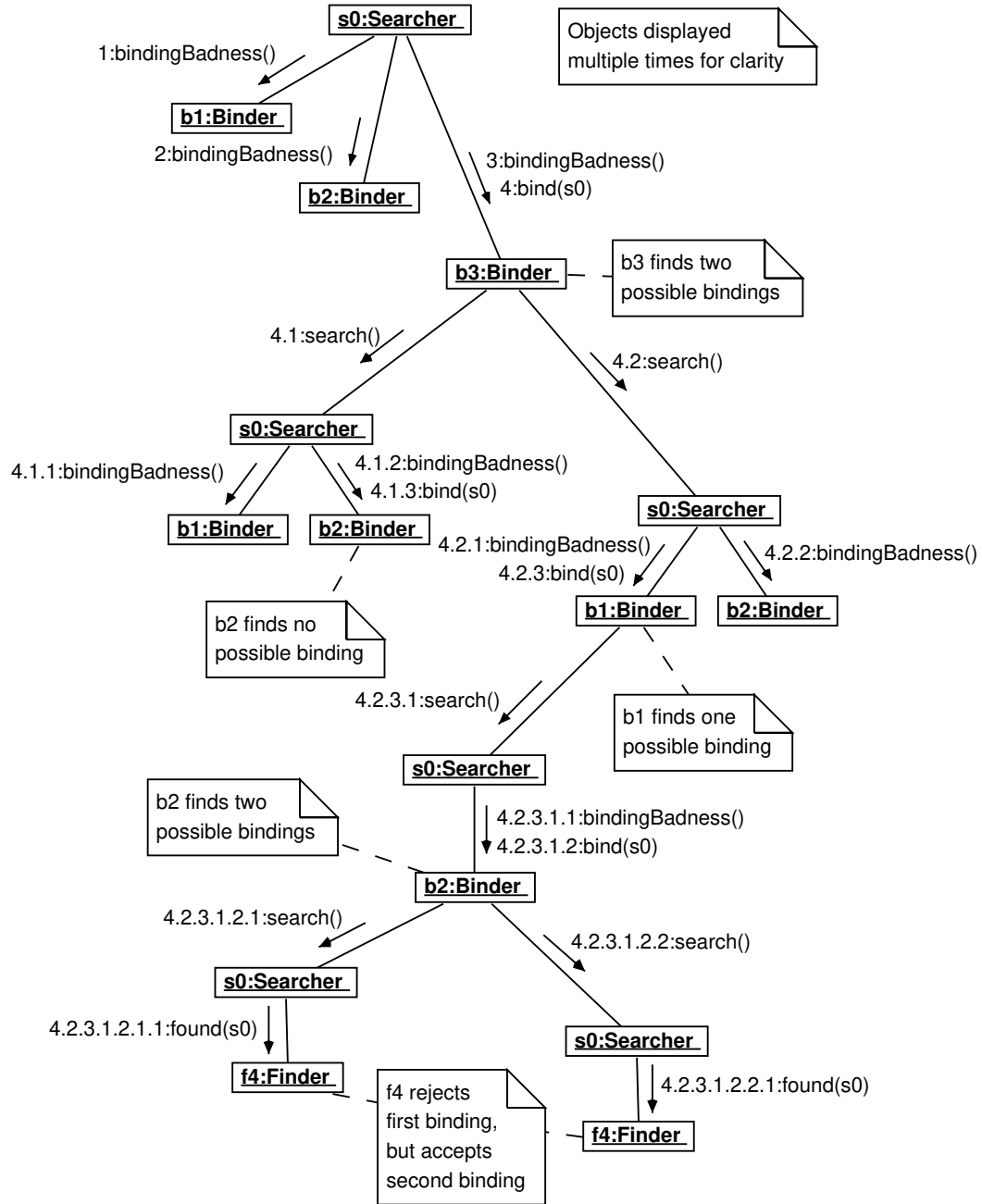


Figure 3.13: A search process

For normal Petri nets a firing mode must only consider the bindings of all variables. We want to manage nets with synchronous channels, too, so that several transitions can be taken part in a firing. In particular, only during the search procedure, when variables are gradually bound, it becomes clear, which transitions need to be synchronised. This cannot be assumed beforehand.

Therefore it is necessary that the searcher keeps track of the involved transition instances during the search procedure. Alongside with the involved transition instances a record of variable bindings has to be administered, hence it seems natural to aggregate pairs of one transition instance and the record of variables in a class. This combination is called transition occurrence and managed by the class **TransitionOccurrence**.

An interface **Occurrence** abstracts from the concrete characteristics of a transition occurrence and can be used to integrate occurrences of active objects of other formalisms.

Whenever a transitions instance is to be checked for enabledness and whenever it is selected as target of a synchronisation, a transitions occurrence is produced by it, and this occurrence enters into a set of occurrences with the searcher.

At the same time inscription occurrences, objects implementing **InscriptionOccurrence**, of all transition inscriptions of the transition in question, are produced. With inscription occurrences the concept that the semantics of transitions results from aggregated inscriptions, is repeated on the dynamic level.

Each inscription occurrence administers necessary information of this inscription during the search procedure, for example the allocation of used variables. Alone the transition occurrence knows all used variables. The inscription occurrences are responsible for the production of binders if they are required by the inscription.

A transition instance creates an occurrence of itself while **startSearch(Searcher)** is called. It registers itself at a Searcher and automatically asks the occurrence for the binders produced by the inscription occurrences.

The search algorithm administers no inscription occurrences. It will only note the occurrences, which again encapsulate the inscription occurrences. This has the advantage, that formalisms for which the active elements are not composed by individual inscriptions can be also treated by this simulation algorithm. In addition, this secures a maximum independence from the individual occurrences.

inscription	occurrence	binder(s)
<b>ActionInscription</b>	<b>ActionOccurrence</b>	N/A
<b>Arc</b>	<b>ArcOccurrence</b>	<b>InputArcBinder</b> <b>TestArcBinder</b> <b>InhibitorArcBinder</b> <b>ArcAssignBinder</b>
<b>ClearArc</b>	<b>ClearArcOccurrence</b>	N/A
<b>ConditionalInscription</b>	<b>ConditionalOccurrence</b>	<b>ConditionalOccurrence</b>
<b>CreationInscription</b>	<b>CreationOccurrence</b>	N/A
<b>DownlinkInscription</b>	<b>DownlinkOccurrence</b>	<b>ChannelBinder</b>
<b>EnumeratorInscription</b>	<b>EnumeratorOccurrence</b>	<b>EnumeratorBinder</b>
<b>FlexibleArc</b>	<b>FlexibleArcOccurrence</b>	<b>FlexibleArcBinder</b>

Table 3.1: Inscription classes and associated occurrence classes

In Table 3.1 we summarize the different inscription occurrences and the executables that they can create. The **ArcOccurrence** class is special in the sense that it creates different binders depending on its attributes. It may also create more than one binder. If the arc can contribute to the binding information, it will create an **ArcAssignBinder**, which is responsible for trying all tokens in order to produce a variable assignment. In any case, it creates a binder that checks for the availability of the token that is computed by the arc's expression.

### 3.9 Finders

The concept of a finder was already introduced earlier, but we will now investigate the implemented subclasses which link the search process to the execution algorithm. The following subclasses are implemented:

**AbortFinder** This finder aggregates another finder and is responsible for terminating the current search, if so requested.

**EnablednessFinder** This finder terminates the search as soon as the first enabled binding was found, because it is only supposed to determine whether a transition is enabled.

**ExecuteFinder** This finder must initiate the execution of some binding of the transition. For this task, too it is enough to search till the first binding. The finder must, however, make additional records on the found binding.

**CollectingFinder** This finder looks up all possible bindings of a transition and always requires a full search. It is useful if we want to display all enabled bindings of a transition or if we want to select a binding from the set of all bindings in a fair way. The collecting finder can naturally be combined with the abort finder, in order to achieve an abort of the search early. Of course this means that the originally desired information will not be available.

The use of an **ExecuteFinder** in the current form cannot guarantee that all bindings of a transition are found with the same probability. Although different tokens are checked in random order, the order of handling of the input arc binders may influence the probability of certain bindings.

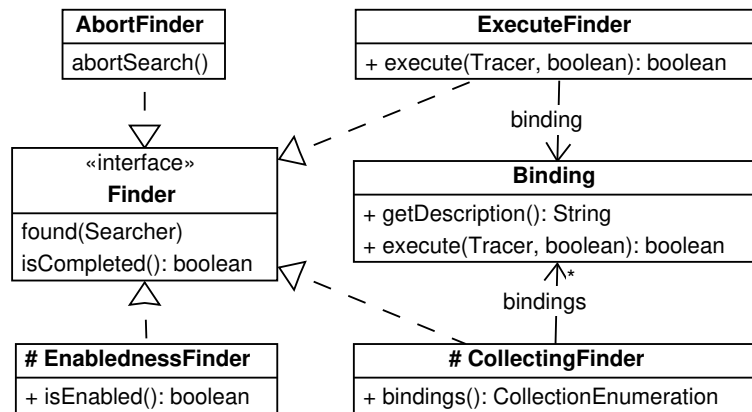


Figure 3.14: The **Finder** classes

Nevertheless all bindings remain firable, so that each possible action in a simulation is possibly observable. Whether a preference in realistic nets occurs or is significant, would have to be determined. We can take care of fairness for all firing modes within a transition using a **CollectingFinder**.

### 3.10 Enacting a Binding

After the search for an enabled binding has been successfully completed, a **Binding** object is created and handed to the finder, as already indicated in Fig. 3.14. The binding object knows about all transition occurrences that are involved in the firing.

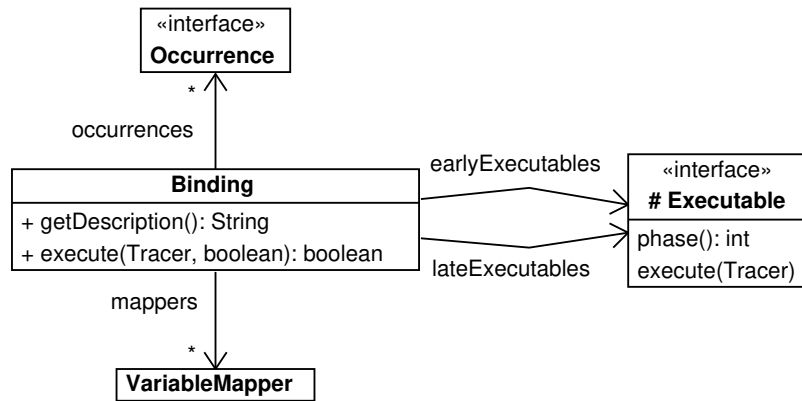


Figure 3.15: The class **Binding**

For each occurrence, the state of the variable mapper immediately after the successful search is preserved in a copy of that variable mapper. All unifiable objects in the mapper are copied, too, in order to protect them from backtracking. These mappers are used to determine the string representation of the binding, if the binding is to be presented to the user.

Separate from that, the binding keeps two sets of executable objects. These objects are generated by the inscription occurrences that were described in Section 3.8. The executable objects come in two flavors: early and late executables.

The interface **EarlyExecutable** is intended for those effects of a transition that may turn out to be impossible if the current marking changes currently to the search process. Such executables are required to be reversible, i.e., they must support a rollback operation.

The other interface is **LateExecutable**, which may contain irreversible effects. However, a late executable may not report an error in any case, it must always complete.

In Table 3.2 you can see a summary of all occurrences and the executables that they create. Most executables depend on the binding of certain variables. If the exact valuation of the variable may not be determined during the creation of the executable, the executable references a variable of the unification algorithm, as witnessed by the class **OutputArcExecutable**. All variables are copied before recording them in the executable object, because backtracking must be eliminated. The same copier is used for all executables, so that executables that bind variable, e.g. the **ActionExecutable**, may pass their results to other executables. The common copier is not reused for the storage of the variable mappers in the binding, because we do not want the string representation of a binding to change.

Fig. 3.16 gives a class diagram of all subclasses of **Executable** and the main related classes.

An **UntestArcExecutable**, which removes the test status from a token, references that executable that originally tested the token. This ensure that the token can be put back using the correct time stamp, if a timed simulation is intended.

Before the executability of early executables is checked, all early executables are locked. The executables may request a certain lock order to avoid deadlocks. Afterward, the check for executability can be performed without disturbances.

Note that the **EarlyConfirmer** does not actually have to lock anything, because it only prints trace messages and it does so only if all other executables succeeded. It cannot even fail, but nevertheless, this executable is considered early, because it must be executed before the first trace messages about removed tokens are printed.

The **FiringStartExecutable** and **FiringCompleteExecutable** are not created by inscription occurrences, but by the firing transition instance itself. They ensure that an

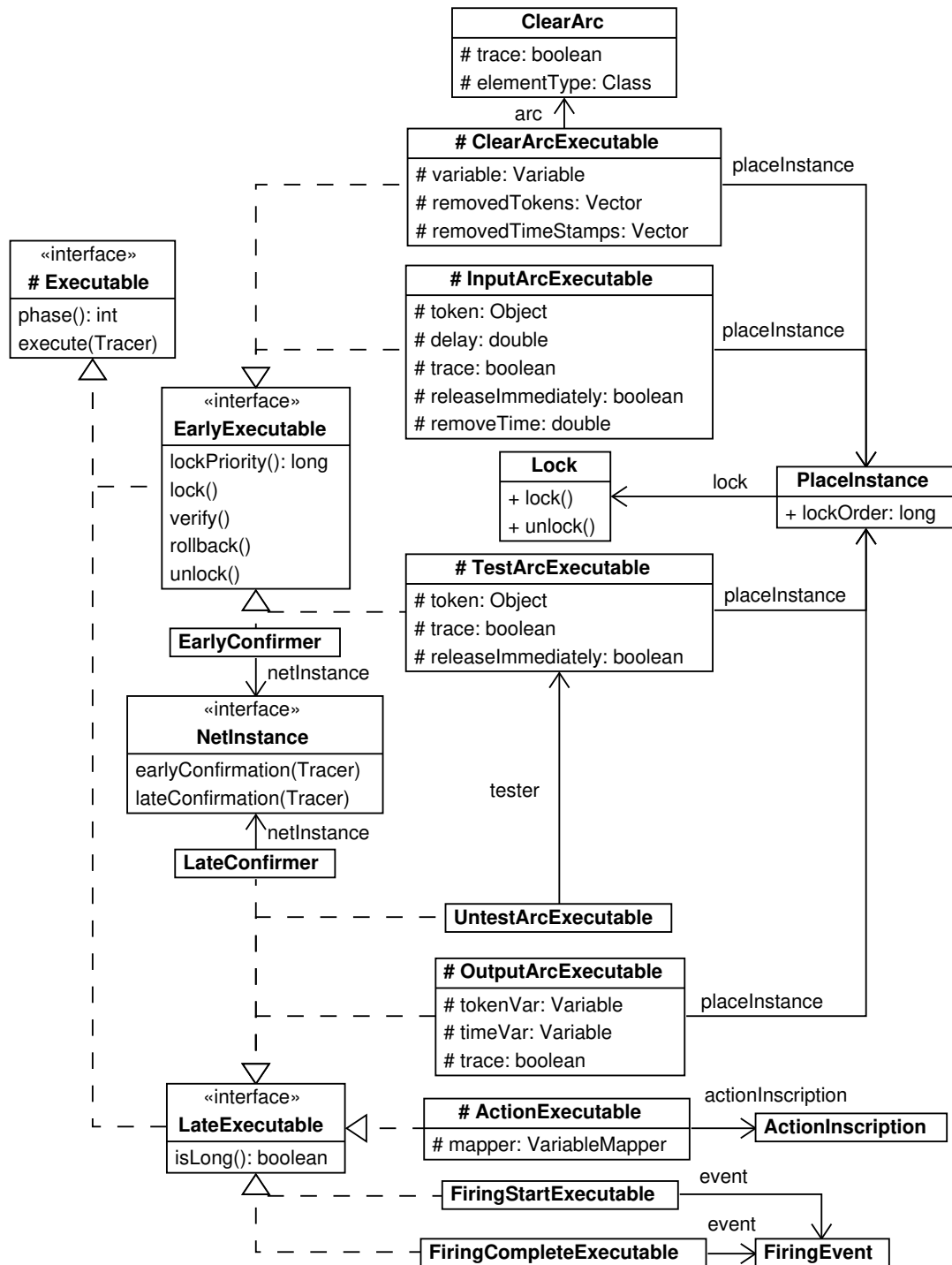


Figure 3.16: The Executable classes



occurrence	case	executable
ActionOccurrence		ActionExecutable
ArcOccurrence	in	InputArcExecutable
	out	OutputArcExecutable
	in/out	InputArcExecutable
		OutputArcExecutable
	fast in/out	InputArcExecutable
	test	TestArcExecutable
		UntestArcExecutable
	fast test	TestArcExecutable
	inhibitor	InhibitorExecutable
ClearArcOccurrence		ClearArcExecutable
ConditionalOccurrence		N/A
CreationOccurrence		EarlyConfirmer
		LateConfirmer
DownlinkOccurrence		N/A
EnumeratorOccurrence		N/A
FlexibleArcOccurrence	in	FlexibleInArcExecutable
	out	FlexibleOutArcExecutable
	fast in/out	FlexibleInArcExecutable

Table 3.2: Occurrence classes and associated executable classes

event is sent to listeners of the transition regarding the start and end of the firing. The **FiringStartExecutable** may be made an early executable, ultimately.

Note that the classes **FlexibleInArcExecutable** and **FlexibleOutArcExecutable** are not represented in the class diagram. They are not essentially different from the other arc classes.

Fig. 3.17 summarizes the life cycle of an early executable object. Note that, if the **verify()** method fails, it is not required to rollback any actions by the executable that failed, but that it is still required to unlock the executable just as all the other executables. All executables whose verification already succeeded must be rolled back, if the binding turns out to be not executable.

## 3.11 The Shadow Layer

In order to separate the simulation engine from the graphical user interface, we added an intermediate data structure that is called a *shadow net system*. A shadow net system abstracts from those information of a net drawing that is irrelevant for the simulation engine. It also allows a uniform interface for the automatic generation of nets that are not supposed to be graphically displayed.

### 3.11.1 Shadow Nets

In Fig. 3.19 you can see the classes that are used to represent a Petri net on the shadow level. No graphical information like size, position, or color is found here, but only topological information, i.e., information about the relationship of transitions, places, arcs, and inscriptions.

A shadow net system consists of an arbitrary number of shadow nets. It also keeps track of a shadow compiler that determines the net formalism used for this shadow net system. At the moment, it is not possible to use different compilers for different nets.

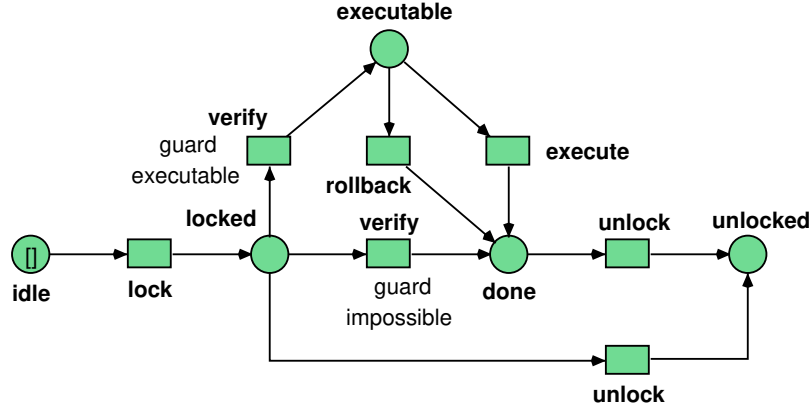


Figure 3.17: The life cycle of an early executable object

A shadow net aggregates a number of shadow net elements, where places and transitions are the most important. Shadow places and shadow transitions are jointly referred to as shadow nodes. Shadow nodes have got a name. All shadow nodes are also inscribable, i.e., they may be annotated by other shadow net elements. Typically, they are annotated by **ShadowInscription** objects.

Inscriptions are not structured further. They are uninterpreted character strings. A shadow inscription may come in two flavors: normal and special. Almost all textual annotations should be normal inscriptions. Special inscriptions may be used, however, if there are different annotation types that cannot be distinguished syntactically. E.g., some net formalisms might use natural numbers for capacities and initial markings alike, so that one inscription type would have to be declared special.

A shadow declaration node is an inscription to the entire net. It is typically used to declare local variables, but this may vary. Typically, there should be at most one declaration node per net.

Shadow arcs are other shadow net elements. They connect places and transitions. They have got a certain arc type: **test**, **ordinary**, **both**, **inhibitor**, **doubleOrdinary**, **doubleHollow**.

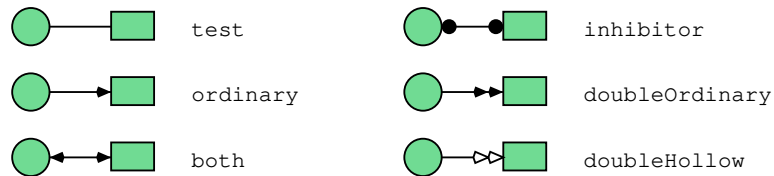


Figure 3.18: The shadow arc types

Every arc type may be assigned a different semantical meaning. Although the names given to the arc types are partially semantic in meaning (e.g. **inhibitor**), they may be interpreted in any way by the net formalism that is realized by the **ShadowCompiler** of the shadow net system.

Arcs like several other shadow objects possess a trace flag that indicates whether a trace message should be printed if this object influences a simulation run. If it is inappropriate to print a trace message, this flag may be ignored.

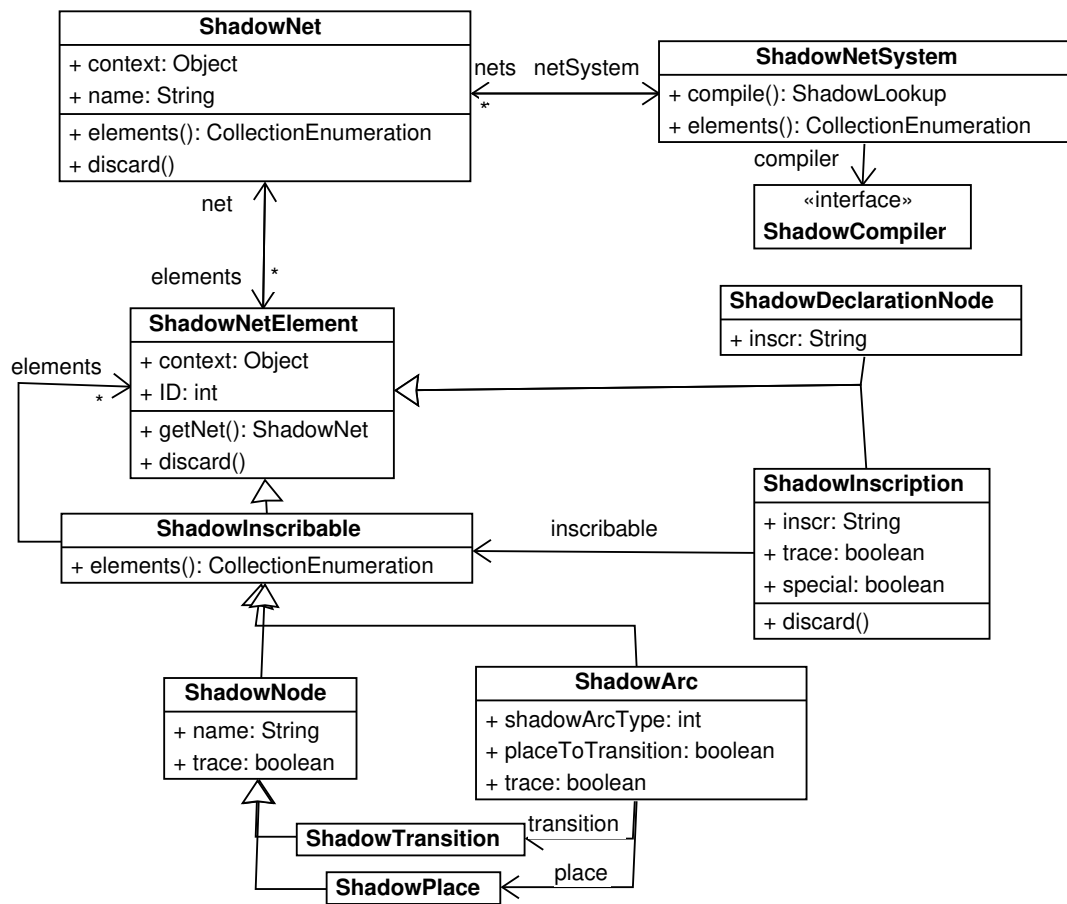


Figure 3.19: The shadow classes

### 3.11.2 Net formalisms

A **ShadowNetCompiler** defines a Petri net formalism. It is responsible for converting a shadow net system into a collection of nets as defined by the simulation algorithm.

In doing this, it needs to create nets, places, and transitions of the semantical level. This mapping need not be one to one, but that can be considered the typical case. The IDs of the shadow net elements can be reused of the IDs of the **Place** and **Transition** elements, so that one can visualize the state of a simulation at the graphical layer.

In doing the transition, it will also be required to parse the textual annotations and to decorate the places and transitions accordingly.

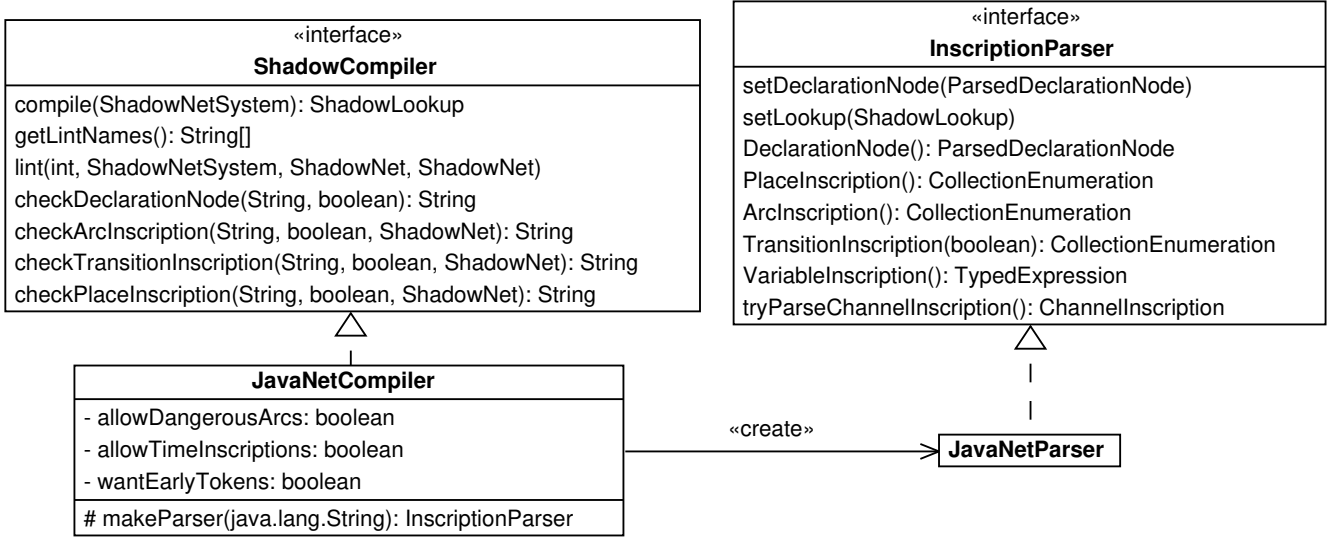


Figure 3.20: The standard shadow compiler

The standard implementation of the shadow compiler is the **JavaNetCompiler** as indicated in Fig. 3.20. It delegates most of the parsing work to an instance of **JavaNetParser**, which implements the interface **InscriptionParser**. For own net formalisms it is suggested to start with **JavaNetCompiler** and subclass it, essentially only overriding the method that creates the parser, because most of the handling of the places and transitions will stay the same for all net formalisms.

Note that the parser is written with JavaCC [3] and that this parser generator does not support inheritance. That means that you might end up copying large parts of your grammar from existing grammars. You might also choose a syntactic niche and modify the standard parser, but make sure that it behaves exactly as before when used with the **JavaNetCompiler**. This might, however, hinder the inclusion of your extension into the main development line.

## 3.12 Simulation

Ultimately, we come to the class that puts together the simulation algorithms that we described so far. It is the **Simulator**, which is responsible for starting and stopping a simulation run. While running, it has to acquire potentially activated transitions from the **SearchQueue**, search for an activate binding using a **Searcher** and fire the **Binding**, if one is found. The main problem that arises at this level is to control the concurrent access to the search queue and to the simulator while starting and stopping.

Fig. 3.21 gives the basic interface of a simulator class and lists the three currently available implementations. Every simulator provides methods to set the desired simulation mode, i.e.,

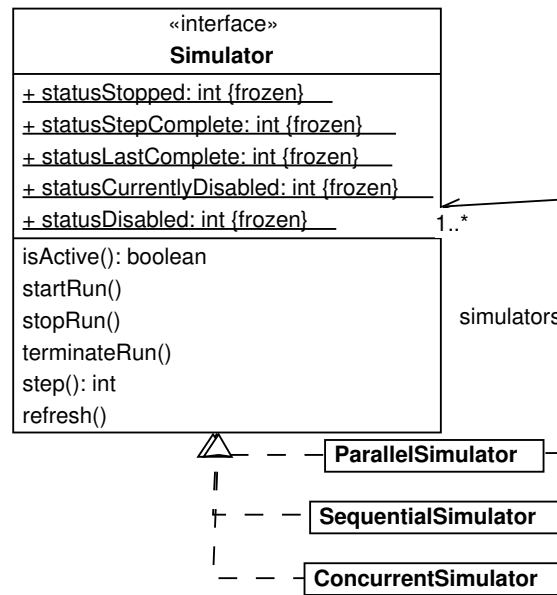


Figure 3.21: The simulator implementations

stopped, single step, or running continuously. A refresh request instructs the simulator to search again for possible binding, even if a prior search failed. This might be required if the user requested a change of the net state during the search process.

After the firing of a step has been requested, the simulator must report in a status code whether a step was actually or not. It must also indicate, whether additional steps might be possible.

status code	Was a transition fired?	Was a transition activated?	Might there be activated transitions in the future?
<b>statusStepComplete</b>	yes	yes	yes
<b>statusLastComplete</b>	yes	yes	no
<b>statusCurrentlyDisabled</b>	no	no	yes
<b>statusDisabled</b>	no	no	no
<b>statusStopped</b>	no	unknown	yes

Table 3.3: The simulator status codes

The **ParallelSimulator** aggregates a number of other simulators that can then search in parallel without knowing of each other. It coordinates the search effort and makes sure that all simulators contribute to a continuous simulation, but only one simulator is requested to perform a single step operation.

Another version is the **SequentialSimulator**, which fires one transition occurrence at a time. This is required for some of the more exotic net formalisms, that are not easily equipped with a partial order semantics. Although this is the simplest implementation of a real simulator, it is already non-trivial.

Synchronisation on the simulator object ensure mutual exclusion of the requests to change the current simulation mode. That means that the only concurrency control has to happen between one thread that performs a continuous simulation and the thread that wants to

change to simulation mode.

The simulation thread is referenced by the field `runThread`. Whenever a new run request is issued, a new thread is created. This allows the threads to be garbage collected upon completion. Reusing one thread would permanently allocate this thread, unless done carefully.

The sequential simulator makes sure to compute another possible binding immediately after it executed another binding. This ensures that the simulator can return a specific status code whenever there are no more activated transitions.

The more elaborated simulator class is named `ConcurrentSimulator`. Here the transitions may be executed concurrently to other transitions and concurrent to a single search process. In fact, the simulator tries to execute transitions as sequentially as possible. If an inscription of a transition could possibly require the firing of other transitions in order to be completed, the simulator will detach the remaining execution of a binding from the search thread. Transitions that do not involve longish actions, however, will be executed synchronously.

Because bindings are sometimes executed concurrently, there might always be firing bindings that can still deposit tokens in certain place instances. Hence it is infeasible to determine that there will be no more activated transitions. Therefore the sequential simulator returns less specific status codes, always indicating that there might be further activated transitions.

Fig. 3.22 shows the basic structure of the concurrent simulator. The thread that controls the simulator, typically the GUI, invokes the transitions at the right. They set the desired mode of operation: termination (-1), stopped (0), single step (1), or continuous run (2). The desired mode in turn influences the simulation thread. Unless a continuous run was requested, the controlling thread will wait until the simulation thread reached an idle state where no more firings are tried.

It is not shown in the Petri net, how the status code is passed between the simulator thread and the controlling thread. This is done using the field `stepStatusCode` in the implementation.

The termination of the search process is shown only schematically. The extended time span that a search requires is indicated by a looping transition that searches and an ultimate transition that finds a binding or determines a dead transition. In the real implementation, the termination request is passed to an instance of `AbortFinder`, which stops the search process if one is running.

The `SearchQueue` has been reduced to the single place `possibly activated` in the net. A notification algorithm implemented in the methods `searchQueueNonempty()` and `registerAtSearchQueue()` is required for a Java implementation. One inhibitor arc is used in the net in order to check whether the search queue is empty. The use of an inhibitor arc already indicates that concurrency must be controlled very carefully here.

In order to prevent deadlocks, the simulator obeys the following locking sequence: synchronize on the simulator object, then synchronize on the search queue, then synchronize on a dedicated object `threadLock`.

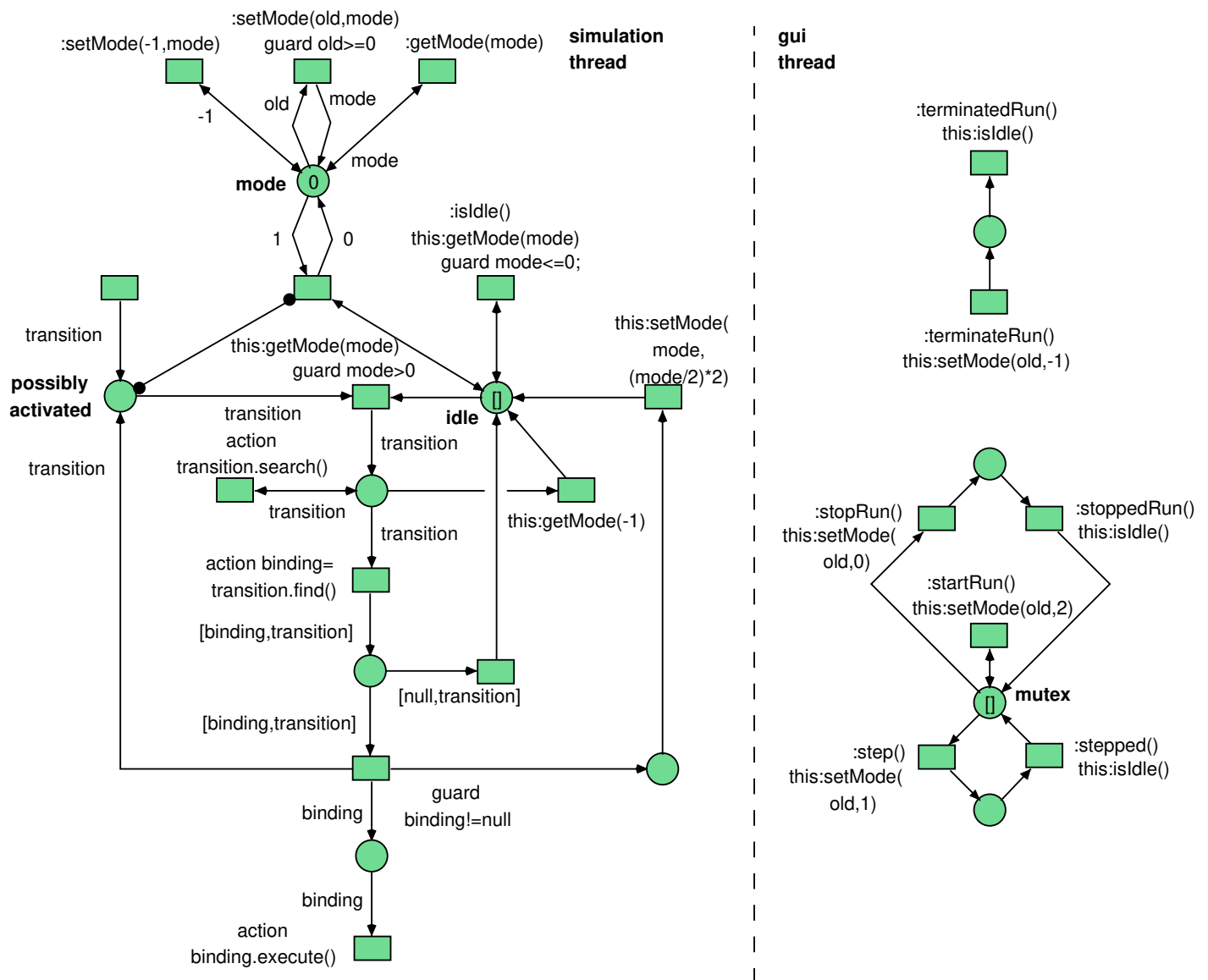


Figure 3.22: The concurrent simulation algorithm as a Petri net

## Chapter 4

# How to Extend Renew

This chapter provides some hints that you might find useful when extending Renew. Some obvious extensions that might have to be introduced are sketched.

### 4.1 Adding a New Arc Type

Renew provides many arc types already, but there are other possible arc types. If you want to add an arc type, have a look at the `DoubleArcConnection` class. This is a special variant of the `ArcConnection` class. It gives a good impression how a new connector shape is created.

You will have to extend the shadow API at least by adding a new arc type constant in `ShadowArc`. Your new figure class must correctly create shadow arcs with the new type.

You must now create a new Renew mode that can interpret your net formalism. Look at `SequentialJavaMode` to see how to add new tools in the `createAdditionalTools(...)` method. Modify the method `getCompiler()` to create a new shadow compiler object for your net formalism.

In the simplest case, this might be subclass of `JavaNetCompiler` that simply overrides the method `getArcFactory(ShadowArc)` by a method that returns an instance of a new subclass of `ArcFactory`.

An arc factory is given an already compiled place and a transition of the static net layer. It is also provided with the type of the place, a flag that indicates whether trace messages should be generated for this arc, and a `TimedExpression`. If a time annotation in the form `expr@time` is present, the timed expression will report `isTimed()` as true. The timed expression can report the expression itself and the time annotation separately, if present.

On the static net layer, arcs are considered special transition inscriptions. The arc factory has to generate one or more inscriptions and add them to the transition that was passed to the factory. You might want to modify the class `Arc` or to create another implementation of `TransitionInscription`.

See the following section on details, how to create transition inscriptions. In that course of events, you might have to extend the functionality of the `TokenReserver` class and possibly that of the `PlaceInstance` class.

### 4.2 Adding New Transition Inscriptions

Unlike arcs, which were discussed in the previous section, textual annotations do not require special figures, so that you can start immediately by creating a new mode and a new compiler class. The compiler, if derived from the `JavaNetCompiler`, can override the method `makeInscriptions(...)`, which takes a single textual inscription and parses it into a collection of semantic inscriptions. Often, this is not even required. Instead, the compiler can



simply provide a specialized parser implementation via the method `makeParser(String)`. This parser can then implement the method `TransitionInscription(...)` accordingly.

The method should return a collection of objects implementing `TransitionInscription`. In some cases, a new inscription is merely a shorthand and can be composed of existing transition inscriptions, but sometimes a specialized implementation has to be generated.

That implementation must be able to build object of the type `TransitionOccurrence` at the beginning of the search for a binding. An occurrence may create binder objects, which can guide the search process by providing binding information. A transition occurrence must also be able to create `Executable` objects, if the search for a binding succeeds.

Input arcs and any transition inscription whose actions can be easily undone will typically require an `EarlyExecutable`, whereas output arcs and inscriptions with irreversible effects will typically require a `LateExecutable`.

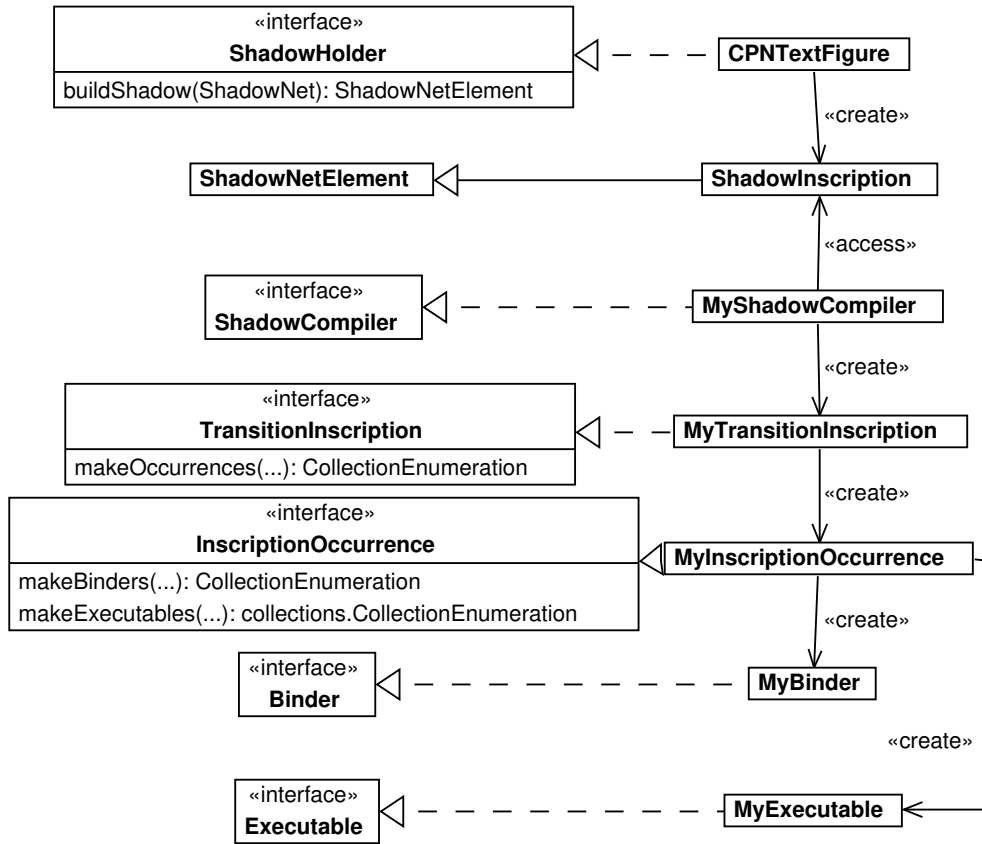


Figure 4.1: Handling of textual inscriptions

Fig. 4.1 summarizes the flow of information through the various classes involved in the translation and execution of a textual transition inscription. Essentially, you see a chain of factories that starts from the graphical figures and ends at the executable objects that are created after the search for a binding succeeded.

### 4.3 Adding a New Inscription Language

By default, Renew uses an inscription language that is very similar to Java. If it is desired to use an entirely different inscription language, a new parser must be written, but the existing arc types and inscriptions types might be sufficient.

You will need a new implementation of **RenewMode**, which you can probably derive from **JavaMode**. That implementation can supply a **ShadowCompiler** of its own, possibly a subclass of **JavaNetCompiler**. The compiler is responsible for converting shadow net into semantical nets.

If you choose to work from **JavaNetCompiler**, you must override **makeParser(String)** and write a parser that can handle your net formalism.

Since the parser need to create transition inscriptions that will typically use **Expression** object, if the net formalism handles colored nets, you need to create expressions or at least implementations of **Function** that supply all operations needed for your net formalism. The parser can then build expressions using the supplied function objects and the **CallExpression** class.

When your net formalism requires unifiable objects or objects that support pattern matching, which are not directly supported by the classes in **de.renew.unify**, you need to create additional implementations of **Unifiable**. Look at **Tuple** for the prototypical unifiable object. You might want to extend the class **TupleIndex** to handle also your additional unifiable objects, so that a fast access to matching tokens in a place becomes possible.

## 4.4 Adding Graphical Figures and Tools

Section 4.1 already contained information about modifications to the existing graphical figures. If you need to add entirely different figures, e.g. for illustration purposes, you should start from **AttributeFigure** as provided by **JHotDraw** and add own drawing routines.

You might want to create a specialized creation tool for your figures, which should be derived from **AbstractTool** or **CreationTool**.

Your figure should provide handles to allow direct manipulations.

## 4.5 Adding Simulation Statistics

The event mechanism of **Renew** as implemented in the package **de.renew.event** is quite suitable to gather statistics about a simulation run. It is possible to use a specialized shadow compiler that compiles a shadow net system by delegating to a different compiler, but sets up event handlers for simulation statistics afterward, when the net is fully compiled.

It is also possible to integrate such support directly in the **Place** and **Transition** classes, but that might be more difficult, although more flexible.

## 4.6 Adding Import and Export Filters

When importing or exporting Petri net formats, you may either program a standalone converter that would typically exploit the XML file format of **Renew**, or you can integrate the converted in **Renew**, which is preferred.

Graphical imports and exports will typically use the **JHotDraw** framework to create or analyse a drawing.

Exports to a non-graphical format can also access the drawings, but they can also convert the nets into the shadow format, which is somewhat simpler to use, and build the export at that level.

Imports from a non-graphical format may either create figures directly or create a shadow net system first and use a **ShadowNetSystemRenderer** afterward to create the actual drawing. In both cases, it should be considered to use the automated net layout to make the nets more readable.

If a certain layout can be inferred from the logical structure of the imported net, then the import method should exploit that information to create figures with the appropriate position directly.

## Chapter 5

# Java Bugs

In this chapter we will describe a few Java bugs that we found especially grievous during our development work. Any hints to alleviate these problems are greatly welcome.

### 5.1 Graphics object loses draw commands

Under Sun/Solaris we observed the following effect with all JDKs: When many hundreds of objects are currently selected, the handles are not correctly drawn on the screen. This is caused by the `Graphics` object that Java passes to the `update(...)` method of the `StandardDrawingView`. It looks as though not all handles of the selected objects are present. We have not found any workaround for this problem.

### 5.2 Packing a frame is not portable

Invoking `Frame.pack()` leads to unpredictable results under some window managers and JDK implementations. Sometimes the windows are reduced to zero width and height.

### 5.3 Window titles are not shown correctly

Under some window managers and JDK implementations the window titles of frames are not correctly displayed.

This bug was recognized, but closed without fix by Sun.

### 5.4 Memory leak through event objects

In some cases, event objects are retained by the AWT event queue even after their processing. This leads to a memory leak.

### 5.5 Memory leak through windows

In some cases, a closed window is retained by the AWT event queue even after its disposal. This leads to a memory leak.

### 5.6 Windows move on the screen

Under some window managers and JDK implementations windows move around the screen unpredictably during certain operations.

## 5.7 Bad fonts and symbols

Using the Sun JDK, the method `Graphics.drawBytes(...)` does not select the correct font unless `Graphics.drawString(...)` is called before. `Graphics.drawString(...)` does not correctly support all font encodings, however.

# Bibliography

- [1] Erich Gamma. *JHotDraw*, 1998.  
Available at <http://members.pingnet.ch/gamma/JHD-5.1.zip>.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [3] JavaCC - The Java Parser Generator, 2000.  
WWW page at <http://www.metamata.com/javacc/>.
- [4] Doug Lea. *Overview of the `collections` Package, Version 0.96b*. State University of New York at Oswego, 1998.  
WWW page at <http://gee.cs.oswego.edu/dl/classes/collections/>.
- [5] Rational. *Unified Modeling Language Resource Center*, 2000.  
WWW page at <http://www.rational.com/uml/>.

# Appendix A

## Glossary

A number of terms are used in this manual to describe the architecture of Renew and its functionality.

**Action Inscription** An  $\rightarrow$  inscription of a  $\rightarrow$  transition that may invoke a complex operation during the  $\rightarrow$  firing of a transition.

**Activated** Synonymous with  $\rightarrow$  enabled.

**Attribute** Any of a number of modifiers for figures in the graphics editor. Most attributes influence the graphical appearance of  $\rightarrow$  figures, but some attributes apply only to the behaviour of a  $\rightarrow$  net during a  $\rightarrow$  simulation.

**Arc** An arrow between a  $\rightarrow$  place and a  $\rightarrow$  transition that indicates the movement of a  $\rightarrow$  token while the  $\rightarrow$  transition  $\rightarrow$  fires. In so far as the arc can be interpreted as an effect of the  $\rightarrow$  transition, arcs are quite more similar to inscriptions.

**Binder** An object that can contribute knowledge about possible bindings of variables during a search. Typically, a binder checks multiple possibilities.

**Binding** A binding characterizes a mode of operation for a  $\rightarrow$  transition. Bindings are typically characterized by associating values to variables. If  $\rightarrow$  synchronous channels are present, a binding is also characterized by the invoked  $\rightarrow$  transition occurrences.

**Bound** An object of the  $\rightarrow$  unification algorithm that is complete and does not contain any  $\rightarrow$  calculator objects.

**Calculator** An object of the  $\rightarrow$  unification algorithm that represents a possible future computation. This is used to represent  $\rightarrow$  action inscriptions during the search for an  $\rightarrow$  activated binding.

**Child** A  $\rightarrow$  figure that must be associated to a  $\rightarrow$  parent figure. Changes to a child do not necessarily affect the  $\rightarrow$  parent.

**Compiler** A compiler converts a  $\rightarrow$  net drawing into a  $\rightarrow$  net. Note that a compiler does not, in this context, need to create machine code or virtual machine code, but that an intermediate representation is sufficient.

**Confirmation** After the creation of a  $\rightarrow$  net instance, the  $\rightarrow$  net instance is not yet fully available. Especially, the transitions are not yet entered into the  $\rightarrow$  search queue. Only the confirmation of a  $\rightarrow$  net instance ensures the full operability.

**Drawing** A collection of  $\rightarrow$  figures which are stored and edited jointly.

**Drawing Context** A drawing context can influence the way in which a  $\rightarrow$  drawing is displayed on the screen. Thus it is possible to display the same drawing in many contexts at the same time.

**Editor** An editor is a program that allows the creation and modification of  $\rightarrow$  drawings. An editor provides  $\rightarrow$  tools and menu commands and provides  $\rightarrow$  views on the drawings.

**Enabled** A  $\rightarrow$  transition instance is called enabled if it can behave in a way that is allowed by the  $\rightarrow$  net formalism. Among the many possible behaviors of a  $\rightarrow$  transition, the enabled behaviors characterized by the means of an enable binding. A single enabled  $\rightarrow$  transition may allow multiple enabling  $\rightarrow$  bindings. Only  $\rightarrow$  spontaneous transitions are said to be enabled.

**Event** Typically a change of the state of an object that must be propagated to all  $\rightarrow$  listeners. Also those special objects that are used as arguments during a notification method call.

**Event Listener** An object that wants to be informed about certain  $\rightarrow$  events. The listener will typically register itself at the  $\rightarrow$  event producer.

**Event Producer** An object that can send an  $\rightarrow$  event to a  $\rightarrow$  listener.

**Executable** An object that encapsulates one effect of a firing transition, e.g., moving a token or executing some code.

**Expression** A formula that can compute values using the assignment of variables. An expression can also have side effects.

**Factory** An object that is responsible for the creation of other objects. See the factory pattern in [2].

**Figure** A graphical object within a  $\rightarrow$  drawing that is characterized by its shape and  $\rightarrow$  attributes. A figure may possess an  $\rightarrow$  ID.

**Fire** An  $\rightarrow$  enabled  $\rightarrow$  transition instance is said to fire if the behavior designated by a  $\rightarrow$  binding is enacted. The firing typically results in the movement of  $\rightarrow$  tokens and sometimes in other changes.

**Guard** A  $\rightarrow$  transition  $\rightarrow$  inscription that computes a boolean condition that may inhibit the transition's  $\rightarrow$  firing.

**Handle** A tiny icon that is associated to a  $\rightarrow$  figure. Handles are only displayed if the current  $\rightarrow$  selection is non-empty. Clicking and possibly dragging a handle modifies the figure in a way that depends on the type of handle.

**ID** An ID, i.e. an identifier, is a value that is associated to an object during its life time. IDs should be unique within a certain domain, but not necessarily globally unique. In fact, if one object is created from a source object, that object may inherit the source's ID. IDs of  $\rightarrow$  figures are natural numbers. Other objects may have more complex IDs.

**Inscription** A textual annotation of a  $\rightarrow$  net or a net element.

**Instance** Each semantics object, e.g. a net or a place, may be instantiated in the same manner that classes are instantiated giving objects. Instances of semantic objects are mutable. Their identity may or may not be externally visible.

**Listener** See  $\rightarrow$  event listener.

**Lock** A lock ensures reentrant mutual exclusion. Reentrant means that one Java thread may access the critical section during recursion.

**Marking** A marking of a place instance or a net instance associates a single place or all places of a net with a  $\rightarrow$  multiset of  $\rightarrow$  tokens.

**Mode** A mode controls the operation of Renew. It specifies the  $\rightarrow$  net formalism used. It may also provide additional  $\rightarrow$  tools and menu entries for editing a  $\rightarrow$  net.

**Multiset** Unlike a set, a multiset may contain a single element more than once. A multiset is typically represented by an assignment of the number of occurrences to each element.

**Net** A template for the creation of  $\rightarrow$  net instances. A net conforms to the semantics of a  $\rightarrow$  net formalism. It is immutable and does not have a state. A net is typically derived from a  $\rightarrow$  net drawing, but it is possible to generate nets non-graphically.

**Net Drawing** A  $\rightarrow$  drawing that represent a  $\rightarrow$  net. The associated  $\rightarrow$  net may depend on the  $\rightarrow$  mode.

**Net Formalism** A net formalism describes the constructs allowed within a  $\rightarrow$  net and their semantics. A net formalism is realized by a  $\rightarrow$  compiler and supported by a  $\rightarrow$  mode in Renew.

**Net Instance** A net instance consists of a  $\rightarrow$  net together with a marking and an identity. There may be many instances of a single  $\rightarrow$  net. Depending on the  $\rightarrow$  net formalism, there may be additional  $\rightarrow$  net elements besides  $\rightarrow$  places,  $\rightarrow$  transitions, and  $\rightarrow$  arcs.

**Net Stub** A special  $\rightarrow$  stub that converts Java method calls to itself to synchronous channel invocations of a  $\rightarrow$  net instance.

**Occurrence** An  $\rightarrow$  instance that is about to become active. An instance may become active multiple times during one step of the simulator, hence more than one occurrence of an  $\rightarrow$  instance may be contained in a  $\rightarrow$  binding.

**Occurrence Check** A part of the unification algorithm that makes sure that no object is part of itself. This has no relation whatsoever with the term *occurrence* at the level of nets.

**Place** A place provides the ability to associate state information with a net. When a distinction is not necessary,  $\rightarrow$  place instances are referred to as places.

**Place Instance** An instance of a  $\rightarrow$  place in a  $\rightarrow$  net instance. At each point of time there is a  $\rightarrow$  marking associated to a place instance.

**Parent** A figure that may contain  $\rightarrow$  child figures. Moving or discarding a parent moves or discards all  $\rightarrow$  children in the same way.

**Search Queue** The search queue keeps track of all possibly enabled  $\rightarrow$  transition instances.

**Searcher** An object that controls the search for an activated  $\rightarrow$  binding of a  $\rightarrow$  transition.

**Selection** In an  $\rightarrow$  editor there may be a set of selected  $\rightarrow$  figures. Typically, these figures were previously accessed by a  $\rightarrow$  tool. Menu commands typically operate on the selected  $\rightarrow$  figures. For a selected  $\rightarrow$  figure, its  $\rightarrow$  handles are displayed.

**Shadow** The shadow layer separates the GUI from the execution layer. Shadow nets represent the net drawings, but they abstract from all information that does not influence the simulation, like position, size, or color of the net elements.

**Simulation** The act of putting the intended behavior of a system of  $\rightarrow$  nets into practice. This is synonymous with execution, which would be the common idiom outside the Petri net world.



**Simulator** A simulator is responsible for controlling a  $\rightarrow$  simulation. Technically, the simulator uses a  $\rightarrow$  searcher to search for activated  $\rightarrow$  bindings and executes them afterward.

**Spontaneous** A transition is said to be spontaneous, if it does not contain an  $\rightarrow$  uplink.

**Synchronous Channel** A synchronous channel connects two two active entities and forces them to operate jointly, e.g. two transitions would have to  $\rightarrow$  fire at the same time. Synchronous channels come in two flavors:  $\rightarrow$  uplinks and  $\rightarrow$  downlinks.

**Strategy** An object that is responsible for the execution of some algorithm. Typically, a strategy is immutable. See the strategy pattern in [2].

**Stub** An object that forwards all incoming calls in a appropriate way to another object.

**Token** An elementary object that is associated to a  $\rightarrow$  place or  $\rightarrow$  place instance by a  $\rightarrow$  marking.

**Tool** An editing procedure for  $\rightarrow$  drawings that typically requires multiple interactions on the side of the user. The currently active tool determines the reaction of the  $\rightarrow$  editor to clicks within a  $\rightarrow$  drawing.

**Transaction** A transaction groups a number of actions into an atomic block. A  $\rightarrow$  binding should execute its effects in a transaction.

**Transition** A transition provides the ability to associate possible behavior with a  $\rightarrow$  net. When a distinction is not necessary,  $\rightarrow$  transition instances are referred to as transitions.

**Transition Instance** An instance of a  $\rightarrow$  transition in a  $\rightarrow$  net instance. Transition instances may be  $\rightarrow$  enabled by a  $\rightarrow$  binding.

**Transition Occurrence** One activation of a  $\rightarrow$  transition instance. A single  $\rightarrow$  transition instance may give rise to multiple concurrent transitions occurrences in the same or in different  $\rightarrow$  bindings.

**Tuple** A tuple is a  $\rightarrow$  unifiable object the aggregates a number of other objects.

**Unifiable Object** An object that is handled by the unification algorithm. All unifiable objects are subject to the  $\rightarrow$  occurrence check.

**Unknown** A tuple is a  $\rightarrow$  unifiable object about which absolutely nothing is known except for its identity. Further unifications may make an unknown complete or even bound.

**Uplink** An uplink constitutes one end of a synchronous channel. A  $\rightarrow$  transition instance with an uplink cannot  $\rightarrow$  fire on its own, but must wait until the uplink is invoked by a different transition.

**Uplink Provider** An object that owns one or more  $\rightarrow$  uplinks that can be accessed via a  $\rightarrow$  synchronous channel.

**View** A view displays a  $\rightarrow$  drawing or a part thereof. One  $\rightarrow$  drawing may possibly be shown in multiple views, perhaps using different  $\rightarrow$  drawing contexts.