

# Renew – User Guide

Olaf Kummer

Frank Wienberg

Michael Duvigneau

Lawrence Cabac

Michael Haustermann

David Mosteller

Marcel Hansson

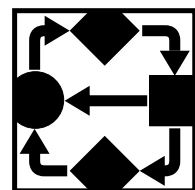
University of Hamburg

Department for Informatics

Theoretical Foundations Group

Release 5.0-Pre

January 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/renew.pdf` as distributed together with Renew 5.0-Pre.



# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Should I Use Renew? . . . . .	8
1.2	How to Read This Manual . . . . .	9
1.3	Acknowledgements . . . . .	9
<b>2</b>	<b>Installation</b>	<b>11</b>
2.1	Prerequisites . . . . .	11
2.2	Possible Collisions . . . . .	11
2.3	Upgrade Notes . . . . .	11
2.3.1	General . . . . .	11
2.3.2	Upgrade from Renew 1.5 or earlier . . . . .	12
2.3.3	Upgrade from Renew 1.6 or earlier . . . . .	12
2.3.4	Upgrade from Renew 2.0/2.0.1 or earlier . . . . .	12
2.3.5	Upgrade from Renew 2.1/2.1.1 or earlier . . . . .	12
2.3.6	Upgrade from Renew 2.2 or earlier . . . . .	13
2.3.7	Upgrade from Renew 2.3 or earlier . . . . .	13
2.3.8	Upgrade from Renew 2.4.3 or earlier . . . . .	13
2.3.9	Upgrade from Renew 2.5 or earlier . . . . .	13
2.3.10	Upgrade from Renew 2.5.1 or earlier . . . . .	13
2.3.11	Upgrade from Renew 2.6 or earlier . . . . .	13
2.3.12	Upgrade from Renew 4.1 or earlier . . . . .	13
2.4	Installing Renew . . . . .	13
2.4.1	Base Installation . . . . .	14
2.4.2	Source Installation . . . . .	14
2.5	Platform-specific Hints . . . . .	14
2.5.1	MacOS . . . . .	15
2.5.2	Unix . . . . .	15
2.5.3	Windows . . . . .	16
2.6	Special Configuration Options . . . . .	16
2.6.1	Ways of configuring Renew . . . . .	16
2.6.2	Drawing Load Server . . . . .	18
2.6.3	Multiprocessor Mode . . . . .	18
2.6.4	Sequential Mode . . . . .	19
2.6.5	Class Loading (and Reloading) . . . . .	19
2.6.6	Net Loading . . . . .	20
2.6.7	Database Backing . . . . .	21
2.6.8	Remote Simulation Access . . . . .	22
2.6.9	Logging . . . . .	23
2.7	Plug-ins . . . . .	24
2.7.1	Install Plug-ins . . . . .	24
2.7.2	Exclude Plug-ins Temporarily . . . . .	24
2.7.3	System Termination . . . . .	25



2.7.4	Commands . . . . .	25
2.7.5	Console . . . . .	27
2.7.6	Net Components . . . . .	28
2.8	Troubleshooting . . . . .	29
2.9	History . . . . .	29
<b>3</b>	<b>Reference Nets</b>	<b>30</b>
3.1	Net Elements . . . . .	30
3.2	I do not Want to Learn Java . . . . .	32
3.3	A Thimble of Java . . . . .	32
3.4	The Inscription Language . . . . .	35
3.4.1	Expressions and Variables . . . . .	35
3.4.2	Types . . . . .	36
3.4.3	The Equality Operator . . . . .	37
3.4.4	Method Invocations . . . . .	38
3.5	Tuples, Lists, and Unification . . . . .	39
3.6	Net Instances and Net References . . . . .	40
3.7	Synchronous Channels . . . . .	41
3.8	Manual Transitions . . . . .	44
3.9	Calling Nets from Java . . . . .	45
3.9.1	Net Methods . . . . .	45
3.9.2	Event Listeners . . . . .	47
3.9.3	Automatic Generation . . . . .	48
3.10	Additional Arc Types . . . . .	52
3.10.1	Flexible Arcs . . . . .	52
3.10.2	Clear Arcs . . . . .	54
3.10.3	Inhibitor Arcs . . . . .	55
3.11	Timed Nets . . . . .	56
3.12	Pitfalls . . . . .	58
3.12.1	Reserve Arcs and Test Arcs . . . . .	58
3.12.2	Unbound Variables . . . . .	58
3.12.3	Side Effects . . . . .	59
3.12.4	Boolean Conditions . . . . .	59
3.12.5	Custom Classes . . . . .	59
3.12.6	Net Stubs . . . . .	59
3.12.7	Execution of <code>synchronized</code> Java Code . . . . .	60
3.12.8	Case of Class and Variable Names in Untyped Nets . . . . .	60
<b>4</b>	<b>Using Renew</b>	<b>61</b>
4.1	Basic Concepts . . . . .	61
4.1.1	Window Management . . . . .	62
4.2	Tools . . . . .	63
4.2.1	The Selection Tool . . . . .	63
4.2.2	Drawing Tools . . . . .	64
4.2.3	Net Drawing Tools . . . . .	70
4.3	Menu commands . . . . .	73
4.3.1	File . . . . .	73
4.3.2	Edit . . . . .	79
4.3.3	Layout . . . . .	81
4.3.4	Attributes . . . . .	84
4.3.5	Net . . . . .	87
4.3.6	Simulation . . . . .	92
4.3.7	Windows . . . . .	98
4.3.8	System . . . . .	98



4.3.9	UI Scale . . . . .	98
4.3.10	Additional Top-Level Menus . . . . .	99
4.4	Net Simulations . . . . .	99
4.4.1	Net Instance Windows . . . . .	99
4.4.2	Current Marking Windows . . . . .	99
4.4.3	Simulation Control . . . . .	100
4.5	Simulation Server . . . . .	100
4.6	Error Handling . . . . .	102
4.6.1	Quick Fix . . . . .	102
4.6.2	Parser Error Messages . . . . .	102
4.6.3	Early Error Messages . . . . .	102
4.6.4	Late Error Messages . . . . .	107
<b>A</b>	<b>Contacting the Team</b>	<b>110</b>
<b>B</b>	<b>File Types</b>	<b>111</b>
<b>C</b>	<b>Keyboard Shortcuts</b>	<b>112</b>
<b>D</b>	<b>License</b>	<b>115</b>
D.1	Contributed Parts . . . . .	115
D.1.1	The <code>collections</code> Package . . . . .	115
D.1.2	The <code>JHotDraw</code> Package . . . . .	115
D.1.3	Code Generated from <code>JavaCC</code> . . . . .	115
D.1.4	Bill's Java Grammar . . . . .	116
D.1.5	Graph Layout Algorithm . . . . .	116
D.1.6	The <code>Reload4j</code> Package . . . . .	117
D.1.7	The <code>FreeHEP VectorGraphics</code> package . . . . .	118
D.1.8	<code>JLine2</code> . . . . .	118
D.1.9	<code>Commons CLI</code> . . . . .	118
D.1.10	<code>DockingFrames</code> . . . . .	118
D.1.11	Other Libraries . . . . .	118
D.2	Original Parts . . . . .	118
D.2.1	Example Nets . . . . .	119
D.2.2	Java Source Code and Executables . . . . .	119
D.3	Created Parts . . . . .	119
D.4	Disclaimer . . . . .	119
D.5	Open Source . . . . .	119



# List of Figures

3.1	The net <b>elements</b>	30
3.2	The net <b>colored</b>	31
3.3	The Java type hierarchy and the hierarchy of lossless conversions	33
3.4	The net <b>gcd</b>	35
3.5	The net <b>gcdtyped</b>	36
3.6	The net <b>equality</b>	37
3.7	The net <b>frame</b>	38
3.8	The net <b>socks</b>	39
3.9	The net <b>reverse</b>	40
3.10	The net <b>creator</b>	41
3.11	The net <b>othernet</b>	41
3.12	The net <b>synchro</b>	41
3.13	The net <b>multi</b>	43
3.14	The net <b>param</b>	43
3.15	The net <b>santa</b>	43
3.16	The net <b>bag</b>	43
3.17	The net <b>mutex</b>	44
3.18	The net <b>account</b>	45
3.19	The net <b>customer</b>	47
3.20	The net <b>buttonmaker</b>	49
3.21	The net <b>sizechanger</b>	49
3.22	The net <b>enumbag</b>	51
3.23	The net <b>enumsanta</b>	51
3.24	The net <b>flexible</b>	53
3.25	The net <b>election</b>	54
3.26	The net <b>visualelection</b>	55
3.27	The net <b>juggler</b>	56
3.28	The net <b>filetypes</b>	56
3.29	The net <b>port</b>	57
3.30	The net <b>reserve</b>	58
3.31	The net <b>buffer</b>	59
4.1	The Renew Window	61
4.2	The Petri Net Toolbars in the Renew main Window	63
4.3	The Renew Navigator	74
4.4	The Attribute bar.	84
4.5	An Example of Browsing Token Objects in Expanded Tokens Mode.	89
4.6	The net <b>iconsanta</b>	90
4.7	The net <b>iconbag</b>	90
4.8	The Santa Claus Example with Icons During Simulation.	91



# List of Tables

2.1	The startup and stub scripts. . . . .	15
2.2	Properties to configure database backing . . . . .	21
2.3	Properties to configure a remote simulation . . . . .	22
3.1	The primitive data types of Java . . . . .	32
3.2	Java binary operators, rules separate operators of equal precedence . . . . .	34
3.3	Java unary operators . . . . .	34
4.1	Summary of selection tool operations . . . . .	65



# Chapter 1

## Introduction

On the following pages, you will learn about Renew, the Reference Net Workshop. The most important topics are:

- installing the tool (Chapter 2),
- the reference net formalism (Chapter 3),
- using Renew (Chapter 4),

Both reference nets and their supporting tools are based on the programming language Java. To be able to use them to their full capacity, some knowledge of Java is required. While the basic concepts of Java will be explained in this document, there are plenty of books and Websites that will serve as a more in-depth introduction to Java.

If you encounter any problem during your work with Renew, we will try to help you. See Appendix A for our address. At the same address, you can make suggestions for improvements or you can request information on the latest release of Renew. If you want to submit example models or extensions to the tool, that would be especially welcome.

### 1.1 Should I Use Renew?

The main strength of Renew lies in its openness and versatility.

- Renew has been written in Java, so it will run on all major modern operating systems without changes.
- Renew comes complete with source, so its algorithms may be freely extended and improved. It is in fact possible to add special net inscriptions quickly. It is even possible to implement completely new net formalisms without changing the basic structure of Renew.
- Renew can make use of any Java class. Today there exist Java classes that cover almost all aspects of programming.
- Reference nets are themselves Java objects. Making calls from Java code to nets is just as easy as to make calls from nets to Java code.

The Petri net formalism of Renew, too, might be very interesting for developers.

- Renew supports synchronous channels. Channels are a powerful communication mechanism and they can be used as a reliable abstraction concept.
- Net instances allow object-oriented modeling with Petri nets. While a few other net formalisms provide net instances, it is their consistent integration with the other features that makes them useful.
- Reference nets were specifically designed with garbage collection of net instances in mind, which is indispensable for good object-oriented programming.



- Many arc types are available that cover almost all net formalisms. Simulation time with an earliest firing time semantics is integrated.

There are, however, a few points to be aware of.

- There are currently only rudimentary analysis tools for Renew. Although a few export interfaces have already been implemented, useful analysis seems a long way off. Currently, Renew relies entirely on simulation to explore the properties of a net, where you can dynamically and interactively explore the state of the simulation. However, for many applications, analysis does not play a prominent role. Petri nets are often used only because of their intuitive graphical representation, their expressiveness, and their precise semantics.
- During simulation, the user cannot change the current marking of the simulated net except by firing a transition. This can make it somewhat more difficult to set up a desired test case.
- In our formalism, there is no notion of firing probabilities or priorities. By exploiting the open architecture of Renew, these features may be added later on, possibly as third-party contributions.
- Renew is an academic tool. Support will be given as time permits, but you must be aware that it might take some time for us to process bug reports and even more time to process feature requests.

But since Renew is provided with source code, you can do many changes on your own. And your feature requests have a high probability to be satisfied if you can already provide an implementation.

## 1.2 How to Read This Manual

It is generally recommended to read all chapters in the order in which they are presented. However, when somebody else has installed Renew for you, you should skip Chapter 2 entirely.

**Renew** If you are already familiar with a previous version of Renew, you should simply skim the manual and look for the Renew 5.0-Pre icons as shown to the left. The paragraphs that are tagged with this icon elaborate on new features of the current version.

**5.0-Pre**

You should also consult Section 2.3 for some notes on the upgrade process. The upgrade might require some explicit actions on your part.

Advanced users may want to consult the architecture guide `doc/architecture.pdf` in the source package of Renew, if it is intended to modify Renew. It is not recommended for the casual user to spend much time reading this manual, as it is quite technical and of little help in the day-to-day use of Renew. The architecture guide is not up to date with the current version of Renew, but may still be helpful to developers.

## 1.3 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 Schuld, 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

# Installation

In this chapter we will give a short overview of the installation process. It is not difficult especially if you are already at ease with the Java environment. But even as a novice you should be able to complete the process successfully.

### 2.1 Prerequisites

You must have Java 17 or higher installed. If you have not done this yet, we suggest that you get the latest Java Development Kit (JDK) from Oracle (<https://www.oracle.com/java/>) or from Adoptium (<https://adoptium.net/>) where versions for Windows, Linux, MacOS are available. Renew only requires the Java Runtime Environment (JRE), but it is no longer distributed separately. All development kits are available free of charge for personal use. We recommend using the version JDK 17 (LTS) or the latest available version.

If you intend to do a source installation, you also need to install a couple of software packages from third parties. See Section 2.4.2 for details.

### 2.2 Possible Collisions

While Renew is based on the JHotDraw package by Gamma [8], the package is distributed with Renew. The package has been substantially improved, so that it is impossible to substitute a different version for it. If you have the original JHotDraw installed, this might result in a problem.

### 2.3 Upgrade Notes

These notes are supposed to help you when you have already installed an earlier version of Renew. In Section 2.9 you can find a list of differences, if you are interested in further details. Keep in mind that for version 5.0-Pre many classes were deprecated, especially one's in the CH plugin. If you are using them, check if a new alternate class was provided and use that one instead, otherwise the remaining deprecations with no current alternatives will be added for the real release of version 5.0.

#### 2.3.1 General

Note that you cannot usually read nets created with a later version by older versions of Renew. However, newer versions of Renew can read older files without problems. I.e., an upgrade to the current version is simple, but irreversible.



### 2.3.2 Upgrade from Renew 1.5 or earlier

Serialized shadow net systems exported by Renew 1.5 or earlier cannot be used with any later versions. You can simply re-export the net system.

A new keyword `manual` was added to the inscription language. In rare cases, you will have to rename a variable or package to account for this change.

### 2.3.3 Upgrade from Renew 1.6 or earlier

The required Java version has changed, you need at least Java 1.4 to run newer versions of Renew.

You do not any longer need to install a separate XML parser (like Xerces) because Renew now uses the built-in parser of Java 1.4.

Saved simulation states exported by Renew 1.6 cannot be used with the current version of Renew.

Stubs compiled with Renew 1.6 or earlier cannot be used with the current version. You can simply recompile the stubs and the resulting Java files.

Several classes of the Renew framework were moved or renamed. In particular, the class `de.renew.simulator.NetInstance` has now become `de.renew.net.NetInstance`. If you use typed variables for net references in your nets, you must adapt the variable declarations or package imports.

When a new net is created, the `:new()` is no longer implicitly invoked. It is only invoked when you create the net using the notation `n:new NetName()` as opposed to `n:new NetName`. You might have to rewrite some nets as a result of this change.

The way of including custom classes in a Renew simulation has changed, please read Section 2.6.5 for more details.

### 2.3.4 Upgrade from Renew 2.0/2.0.1 or earlier

The main application class has been changed from `de.renew.plugin.PluginManager` to `de.renew.plugin.Loader`. There are now two configuration file locations: first, the per-installation configuration file `renew5.0-Pre/config/renew.properties` is read, then the user-specific file `~/.renew.properties` is consulted. The property `pluginLocations` has been changed to a classpath-like syntax (see Section 2.7.1).

The net loader priorities have changed: instead of preferring `.sns` files over `.rnw` files regardless of the order of their directories in the `de.renew.netPath` property, now the order of directories is considered first, while the order of file types is undetermined.

The Renew code has been modified to compile without warnings under Java 1.5, but we still use Java 1.4.2. The conformity with Java 1.5 also implies the use of an up-to-date version of JavaCC during compilation. The old `collections` package has been removed from all components except FS. The package is no longer distributed with the base archive, but included in the FS plug-in.

### 2.3.5 Upgrade from Renew 2.1/2.1.1 or earlier

The required Java version has changed, you need at least Java 1.5 to run newer versions of Renew.

Saved simulation states exported by Renew 2.1.1 or earlier cannot be used with the current version of Renew. The Drawing Load Server (see 2.6.2 now accepts connections from the local loopback interface only (this has been introduced as a security measure).

The JUnit test classes that were sparsely scattered in the Renew code have been migrated from JUnit 3.x to JUnit 4.x architecture.



### 2.3.6 Upgrade from Renew 2.2 or earlier

The required Java version has changed, you need at least Java 6 to run newer versions of Renew.

### 2.3.7 Upgrade from Renew 2.3 or earlier

Files exported to the experimental `.xrn` format cannot be used anymore. Its support has been discontinued. We encourage the use of PNML, instead.

### 2.3.8 Upgrade from Renew 2.4.3 or earlier

The logging configuration files and the default logging directory moved from the home folder to a subdirectory `.renew` in the home folder. In order to use your old configuration, you should move the `.log4j.properties` file to the `.renew` subdirectory in your home folder (you may need to create the directory first). For the logs you now have to look in `.renew/renewlogs` in your home folder (if not configured differently).

### 2.3.9 Upgrade from Renew 2.5 or earlier

The required Java version has changed, you need at least Java 11 to run newer versions of Renew.

### 2.3.10 Upgrade from Renew 2.5.1 or earlier

The PNML export and import were reimplemented because the old implementation based on a prefinal standard of PNML. The new version implements the standard in version 2009 from <http://www.pnml.org/>. PNML files created with earlier versions may produce errors in Renew 5.0-Pre during import. If you encounter such problems, save your files as `.rnw` with the old version to import them in Renew 5.0-Pre.

### 2.3.11 Upgrade from Renew 2.6 or earlier

The new build system used is Gradle and not Apache Ant, furthermore it is recommended, but not required, to have Java 17 to work with Renew. If you want to easily start up Renew, the new command line to do so is `java -p ".:libs" -m de.renew.loader gui` on Linux or `java -p ".;libs" -m de.renew.loader gui` for windows.

### 2.3.12 Upgrade from Renew 4.1 or earlier

The required Java version has changed, you need at least Java 17 to run newer versions of Renew.

## 2.4 Installing Renew

The two zip-files `renew5.0-Prebase.zip` and `renew5.0-Preresource.zip` form the standard Renew distribution. While the former file contains all files that are required for the operation of Renew, the latter file includes the source files, which are generally *not* needed unless you intend to modify Renew or learn about its algorithms. In addition to the base distribution, we provide some plug-ins at our web page.



### 2.4.1 Base Installation

In the following, we assume Unix filename conventions, i.e., directories separated by / (slash). For other operating systems you might need to change it to \ (backslash). Also, the list separation character differs: In Unix-based environments, : is used, while on Windows, the ; is reserved for drive letters, so ; is used for lists.

Extract the base distribution to a directory of your choice. A directory **renew5.0-Pre** will be created in the current directory. Documentation files, for example this manual, are placed in the subdirectory **renew5.0-Pre/doc**. The subdirectory **renew5.0-Pre/samples** contains example nets. The functionality of Renew has been decomposed into several plug-ins which reside as **jar** files in the subdirectory **renew5.0-Pre/plugins**. This is also the place where you can add extra plug-ins (see Section 2.7).

The file **renew5.0-Pre/de.renew.loader.jar** is a **jar**-file that could be used to execute Renew e.g., by saying

```
java -p /path/to/renew5.0-Pre:path/to/renew5.0-Pre/libs \
-m de.renew.loader gui
```

if you extracted the **zip**-file into **/path/to**. On Windows the paths start with your drive and ; is used instead of : as the list operator. The **gui** part at the end of the line is a command that tells the loader to start the graphical user interface of Renew.

Note that for Unix and Windows we provide ready-made startup scripts already. They will be generated when Renew is installed. In that case, you do not have to provide a loader command manually, and some classpath-related issues (see Section 2.6.5) are handled more conveniently for most use cases. See Section 2.5 for details.

### 2.4.2 Source Installation

Usually there is no need to do a source installation. If you feel confused by this section, simply skip it.

Extracting the source distribution will put files into the directories **renew5.0-Pre/src** and **renew5.0-Pre/doc**.

Note that you will require a full-fledged Java compiler, not only a Java runtime environment, to compile the source code. If you do not have a Java compiler yet, you do not need the source code.

See the file **renew5.0-Pre/src/README** that lists the software packages and versions you need and gives some hints on how to compile and run the application.

After you have successfully built Renew, you can test your compilation result. Just follow the instructions in the previous section, but replace any reference to **renew5.0-Pre/de.renew.loader.jar** by **renew5.0-Pre/src/dist/de.renew.loader.jar**. The plugins directory is relocated from its distribution position to **renew5.0-Pre/src/dist/plugins**, too.

If you succeed to run the compiled Renew, you can delete the file **renew5.0-Pre/de.renew.loader.jar** and the directory **renew5.0-Pre/plugins** with the original **class**-files. You should then consider to edit the start scripts for your platform, so that they become aware of the file location changes.

Please note that the **clean** target defined in the gradle **build.gradle** file completely removes the **renew5.0-Pre/src/dist** directory tree. Therefore any modifications (like installed plug-ins) made in that directory tree get lost every time you run **gradlew clean**.

## 2.5 Platform-specific Hints

For a few platforms we provide special installation support. Even in these cases you could install Renew as described above, but your task will be easier if you read this section.

The installation script is typically called **installrenew** or similar. Start this script to install Renew. The script will create the actual startup scripts for Renew. You have to start



the one called **renew** or similar to get the basic version of Renew running. Other scripts allow you to load drawings into a running Renew editor or provide help for the generation and compilation of stubs, as summarized in Table 2.1.

In the next subsections we will only describe the usage of the basic script. The other scripts have a similar behavior when it comes to the interpretation of environment variables, while their specific effects are explained in other Sections of this manual.

script name	use
<b>renew</b>	starts the Renew editor
<b>loadrenew</b>	Java drawing load client (see Subsection 2.6.2 for details)
<b>makestub</b> <b>compilestub</b> <b>jcompile</b>	generates a net stub from a Java interface compiles a net stub to Java code wrapper for <b>javac</b> that includes some Renew classes in the <b>CLASSPATH</b> (see Subsection 3.9.1 for details)
<b>renewexport</b>	exports a drawing to an image run as <b>renewexport input.rnw output.pdf</b> where <b>input.rnw</b> is a path to a Renew drawing file and <b>output.pdf</b> the path of the output file. guesses the format from extension (pdf, eps, png, svg)

Table 2.1: The startup and stub scripts.

### 2.5.1 MacOS

We provide a MacOS application bundle, which is installed the usual way by dragging it to the Application folder. The Renew App is not signed, which may result in a gatekeeper error message during startup. You need to explicitly allow the start of Renew in the Security & Privacy Settings (see <https://support.apple.com/en-us/HT202491>). The application is configured in GUI-only mode without console output or prompt capabilities. It also provides Finder integration for Renew drawing files.

Since MacOS is built upon a Unix core, you can alternatively follow the installation directions for Unix. To do so, open the Terminal application which can be found in **/Applications/Utilities**.

We provide an AppleUI plug-in as optional download, which is already included in the application bundle. This plug-in adds rudimentary support for native Mac OS look&feel to Renew. It allows the **cmd-q** (or **apple-q**) shortcut to close the editor properly (with safety questions for unsaved changes) and it slightly modifies the Mac OS menu bar. See Section 2.7.1 on how to install the plug-in.

Shortcuts listed in this manual with the **Ctrl** modifier key should be used on computers running MacOS with the **Cmd** modifier key instead. Some reserved shortcuts differ slightly under MacOS (see Appendix C).

### 2.5.2 Unix

We supply a simple install script at **renew5.0-Pre/bin/unix/installrenew** that will handle the installation on most flavors of Unix. Run that script with

```
cd renew5.0-Pre/bin/unix
./installrenew
```



and it will create the shell scripts `renew`, `loadrenew`, `compilestub`, `makestub`, `jcompile`, and `renewexport` in the same directory (see Table 2.1).

However, you must make sure that `java` can be called with your current setting of the `PATH` environment variable. It is also required that you start the installation script from the `bin/unix` directory, otherwise it cannot find the location of the package.

We distribute some files to support desktop integration in FreeDesktop-based environments such as Gnome. In the directory `renew5.0-Pre/bin/unix/freedesktop`, there is a `README` file that explains how such desktop integration can be achieved. However, desktop integration still requires a manual configuration process. The `installdesktop` and `uninstalldesktop` scripts automatically install and uninstall the desktop integration in the FreeDesktop environment.

### 2.5.3 Windows

For Windows we provide an installation script in the directory `renew5.0-Pre/bin/win` for your convenience. This batch file has to be started from its own directory.

```
cd renew5.0-Pre\bin\win
./installrenew
```

This will create the batch files `renew.bat`, `loadrenew.bat`, `makestub.bat`, `compilestub.bat`, `jcompile.bat`, and `renewexport.bat` in the same directory (see Table 2.1).

However, you must make sure that `java` can be called with your current setting of the `PATH` environment variable. It is also required that you start the installation script from the `bin/win` directory, otherwise it cannot find the location of the package.

Please check your classpath variable for any unquoted whitespace in it because batch scripts will interpret paths with it as two arguments.

The `installrenew` script additionally creates some registry files for the file type association of Renew files (`addregistry.reg` and `removeregistry.reg`). `addregistry.reg` contains the registry entries to associate Renew file types (`.rnw`, `.draw`, `.aip`, `.sns`) with the `loadrenew.bat` script. You can perform this association by double clicking `addregistry.reg`. With `removeregistry.reg` you can remove the corresponding entries from the registry. Admin rights are required to edit registry entries. The update of the icons for the Renew file types may need a restart to take effect.

## 2.6 Special Configuration Options

There are several options that can help to adapt Renew to your specific needs. Usually you should not need to use these options, so it is best to skip this section on the first reading.

### 2.6.1 Ways of configuring Renew

There are at least two ways to configure any of the options mentioned in the following sections. In Java terminology, an option is configured by setting a property to a value.

**At startup.** You can supply property values on the Java command line with the `-D` option when you start Renew (this is the way you could configure previous Renew releases). For example, to configure that Renew uses a sequential simulator instead of the default concurrent one, you can set the property `de.renew.simulatorMode` to the value `-1`. This can be done in Linux and MacOS by starting Renew with the command line:

```
java -Dde.renew.simulatorMode=-1 -p path/to/Renew:path/to/Renew/libs\
-m de.renew.loader gui
```

In Windows, this can be done by starting Renew with the command line:



```
java -D"de.renew.simulatorMode=-1 -p "path/to/Renew;path/to/Renew/\
libs" -m de.renew.loader gui
```

It is important that you do not insert any spaces between the `-D` option, the property, the `=` sign and the value. Also, property names are case sensitive.

This way, you can configure any property for just one run of Renew. However, you lose the simplicity provided by the platform-dependent startup scripts. You will have to enter the Java command line directly, including the full path to `de.renew.loader.jar` and the initial command `gui` for the loader.

**Permanently.** To configure properties permanently, you can create or edit either the file `renew.properties` in the installation directory `renew5.0-Pre/config` or the `.renew.properties` file (note the initial dot!) in your home directory. The former configures the Renew installation, while the latter provides configuration on a per-user basis. The user settings may override installation-wide values. In the following, the text `.renew.properties` refers to both files. A template for a `.renew.properties` file can be found in the `config` directory of your Renew installation. The file contains several out-commented properties.

The location of your home directory depends on the operating system (for Windows, it even differs between versions). If you do not know where it is, just run Renew. The first two lines of the application log (that is usually printed to the Java console) tell you where Renew expects the configuration files.

One note for Windows users: The Explorer will not allow you to create a file name with an initial dot. But you can create such a file via the command line interface (`CMD.EXE`) and then edit it as usual.

In the `.renew.properties` file, you can for example add the line

```
de.renew.simulatorMode=-1
```

and Renew uses the sequential simulation engine by default on every run. Of course, values taken from the `.renew.properties` file can always be overridden by a `-D` command line option at startup.

**At runtime.** A third way for some (not all) of the configuration options is provided by menus or dialogues in the net editor. To stick with our example, the dialog opened by the menu entry `Simulation/Configure Simulation...` contains a checkbox named `Sequential mode` and a box where you can change the engine multiplicity (for details, see Section 4.3.6). These two elements in combination represent the current value of the property `de.renew.simulatorMode` and its value is changed when you press the `Apply` or `OK` buttons.

As a fourth way to set properties, the plug-in system provides a `set` command. This command comes in useful in combination with the `script` command (see Section 2.7.4) or if you installed one of the prompt plug-ins (`Console` or `GUIPrompt`). Although this command allows to set values for any known and unknown property at runtime, it is not guaranteed that the value change has any effect. In fact, many properties are evaluated only once at startup time.

As a rule of thumb, all options that affect the simulation behavior are evaluated at each simulation setup. For example, the command

```
set de.renew.simulatorMode=-1
```

has no effect on a simulation currently active (if there is any). But the next time you start a simulation from within the running application, it will be sequential.

When you configure properties at runtime, any setting of that property from the command line or the configuration file is overridden. Runtime settings are not stored permanently.



## 2.6.2 Drawing Load Server

Many users like to load documents into the corresponding application just by double-clicking the document in the file manager. A typical problem is that this starts a new instance of the whole application. This is extremely nasty when using a Java application, since a new Java Virtual Machine is started every time. To avoid this problem with Renew, we added a simple server function to Renew (the Drawing Load Server). This server function is automatically configured to use with the `loadrenew` script. You probably do not need to do the manual configuration if you just want to pass files into a running Renew instance.

### loadrenew script

You can use the script `loadrenew` provided by the installation process (see Section 2.4) to connect to a running Renew instance and pass over the names of the files to open. The `loadrenew` script starts a regular Renew instance, if the connection to a Drawing Load Server is not possible (e.g. if Renew is not already running). So this script can be used as default command to load Renew. The script uses the port configured in the `.renew.properties` file (property `de.renew.loadServerPort`) or the default port 65111 if the property is not set.

The idea is to associate the `loadrenew` script to the Renew file extensions (dependent on the installed plug-ins but at least `rnw`). When a drawing is double-clicked in the file manager, the client is invoked and can transmit the drawing's file name, which is then received and loaded by the single Renew application. This is really a nice feature, because it offers a rudimentary operation system integration.

### manual configuration

By setting the property `de.renew.loadServerPort` to a port number, you can tell Renew to listen for local requests to load a certain drawing on a TCP/IP socket. Just specify an unused TCP/IP port number, say 65111, either on the command line or in the configuration file (see Section 2.6.1). When Renew is started with this property, the "Drawing Load Server" is set up on the given port and waits for clients to connect and transmit the full filename of a drawing to open (followed by an end-of-line). Changing this property's value at runtime has no effect. The port 65111 is used by default. If you do not want to start a "Drawing Load Server", you can set the property to `-1`.

We provide a generic client application written in Java that can be used as a Drawing Load Client. Our Java client is started by (this example is again given in Unix syntax, the `\` is used to indicate that the three printed lines should be combined into one command line)

```
java -Dde.renew.loadServerPort=65111 -cp "*/plugins/*:libs/*" CH.ifa\
.draw.application.DrawingLoadClient <file name>
```

where 65111 is an example for the port number on which the server is running (if you decided to configure the property in the `.renew.properties` configuration file, the `-D` part of the command can be omitted).

## 2.6.3 Multiprocessor Mode

Renew provides support for shared-memory multiprocessors. Depending on your specific application and hardware, this can significantly speed up the simulation engine. But note that this feature is still experimental and has not been tested across platforms due to lack of funds. It should not be used for critical applications at the present time. We would be very glad to receive experience reports, so that the code can become more stable.

You must set the property `de.renew.simulatorMode` to the number of concurrent simulation threads you want. Note that it will usually be detrimental to performance, if you configure a number of threads that exceeds the number of physical processors. This property



is evaluated each time a simulation is started. It can also be configured in the **Configure Simulation** dialog (see Section 4.3.6).

Before using multiple processors, you should probably try to optimize performance with other means. You should configure Log4j to discard log events (alternatively you can disable the generation of trace events for all or most net elements). You should open only very few net instance windows, so that the graphical representation of the markings does not need to be updated.

If you configure a negative number for the property `de.renew.simulatorMode`, sequential simulators are used instead of the default concurrent one. Although Renew will do that, it is kind of weird to configure multiple concurrent instances of the sequential simulator. Only the special case of exactly one sequential simulator is of use and explained in the next section.

## 2.6.4 Sequential Mode

By setting the property `de.renew.simulatorMode` to the value `-1`, you can request a sequential mode where transition firings are no longer concurrent. There is usually little reason to do so, but sometimes concurrently executed transition inscription might lead to strange results. You can then select the sequential mode to check your nets.

Please note that net methods as described in Section 3.9 cannot be used in this mode.

As you might have noted, the property used to configure sequential mode is the same as for multiprocessor mode. So the configuration and evaluation notes given in the previous section still apply to this property.

If we restrict the Petri net formalism to purely sequential behavior, we can add certain Petri net extensions that were not suitable for a true concurrency formalism. Most notably, we can use inhibitor arcs and clear arcs. These extensions will be described in Subsections 3.10.2 and 3.10.3.

The sequential arc types can now be added to and removed from the tool bar via the menu entry **Simulation/Show sequential-only arcs** (see Section 4.3.6).

If you try to simulate nets that contain sequential-only features and the simulation is configured to use a concurrent engine, you will encounter an error message. In this case, you just need to set the **Sequential mode** in the **Configure Simulation** dialog (see Section 4.3.6) and restart the simulation.

## 2.6.5 Class Loading (and Reloading)

When you are developing an application that consists of Java code and net drawings, or if you want to extend some Renew plug-in by supplying custom Java classes, then you need to make your classes known to Renew. The Java way of doing this is to add your classes to the Java classpath either by setting the environment variable `CLASSPATH` or by providing an appropriate `-classpath` option to the `java` command.

With the plug-in system (see Section 2.7), providing custom classes via the Java classpath option or variable has some issues. First, our suggested startup command line uses the `-jar` option, which simply overrides any supplied classpath (see the Java tool documentation for details). Second, the plug-in system uses its own class loader to find plug-ins. Due to the Java class loader concept, the plug-in class loader hides all plug-in classes from any class that is loaded via the system classpath. This means that Renew can load and use your custom classes from the classpath, but your classes cannot refer to any Renew class. This problem especially affects net stubs – they need to refer the `NetInstance` class of the simulator plug-in. To work around this, the plug-in system provides the property `de.renew.classPath`.

You can set the property `de.renew.classPath` to any value that follows the syntax and semantics of the Java classpath (which depends on the operating system you use). The property is evaluated once at startup time to configure the class loader of the plug-in system (changes at runtime are not recommended by Java's class loader concept). All classes available through `de.renew.classPath` can be used in simulations and as plug-in extensions. However,



if a class with the same qualified name exists in a plug-in and in the `de.renew.classPath`, the plug-in class takes precedence.

For your convenience, the `renew` and `loadrenew` startup scripts automatically transfers the value of the `CLASSPATH` environment variable to the `de.renew.classPath` property at startup. As long as you stick to these scripts, you should be able to stick to the classical method of configuring the `CLASSPATH` variable to include your custom classes and stubs.

When you are developing an application that consists of Java code and net drawings, you might want to modify and recompile your own classes and use them in your nets without restarting Renew. Therefore, we provide a class reloading mechanism to simulations in Renew.

By setting the property `de.renew.classReinit` to `true`, you can request that all user classes that are referenced by a net are reloaded before every compilation of a net. When the class reloading feature is enabled, the `de.renew.classPath` property is re-read at each simulation setup, but changes to the classpath do not affect the plug-in system.

However, there is a nasty caveat with this feature: Even without changing the classpath, you can have two instances of the same class in the Java VM: Both are loaded from `de.renew.classPath`, but one is known to the plug-in system while the other one is known to the current simulation only. These classes are never identical or equal, if they are compared (for example when the `instanceof` operator is applied to an object of the other class)! Note that this mechanism may result in some problems when you access the Java reflection API, too.

This property can also be changed from the **Configure Simulation** dialog (see Section 4.3.6). It is evaluated each time a simulation is set up. If no simulation is running, the graphical editor evaluates it before each compilation.

Because the reloading of classes might affect performance.

## 2.6.6 Net Loading

When you are using many nets referencing each other, you might want to try the net loading mechanism. When nets are missing during a syntax check, compilation or simulation, the engine will look for matching drawing (`.rnw`) or shadow net (`.sns`) files and automatically load, compile and include them into the net system.

The net loader is configured using the property `de.renew.netPath`, where the value is a list of directories specified in the same syntax as the `CLASSPATH` environment variable used by Java. Currently, the net path can comprise directories only, `.jar` or `.zip` files are not supported. Subdirectories of the given directories are not searched, either. The order of directories in the path list matters, the first directory containing a net will be used.

It is possible to specify netpath directories relative to the classpath. This is done by prepending the directory entry with the (reserved) directory name `CLASSPATH`. For example, if you include the directory (in Unix syntax, Windows users should replace the slash by a backslash) `CLASSPATH/nets` in the `de.renew.netPath` property, then the net loader would look for a missing net file throughout all `nets` directories relative to all classpath entries. The searched classpath includes everything from the Java system classpath, all loaded plug-ins, and the `de.renew.classPath` property (see Section 2.6.5), in that order. When they are included in the classpath, `.jar` and `.zip` files are searched, too.

The behavior of the net loader differs depending on the type of file it finds when looking for a net. If it encounters a shadow net system (`.sns`) file, the net will be loaded into the simulation or used for a syntax check, but it will not show up in a drawing window on the screen. Matching shadow net system files must contain a single net with the same name alone. If more than one net is found in the shadow net system file, it is rejected to avoid confusion about the source of compiled nets. There is a command in the menu **File/Export/Export all (single file each)** to generate such single-net shadow net system files for all open drawings (see Subsection 4.3.1).

When the simulation has been started via the **Simulation** menu in the editor (and *only* then!), the net loader will also look for (`.rnw`) files in the `netPath`. If it finds a matching



drawing file, the drawing shows up in an editor window and is immediately compiled into the running simulation.

If a net can be obtained from several different sources at the same time, the net loader takes the first one in the order of the `netPath` entries. If it encounters `.sns` files in the same directory as `.rnw` files with the same name, it is not defined which one gets loaded.

The editor also uses the net loader during simulation to open drawings from `.rnw` files that are needed to display net instances. But you should be aware that the net loading feature comes with a big pitfall: If `.sns` files and `.rnw` files are not kept up-to-date, compiled nets in the running simulation *may* have a different structure than the nets shown in the editor! Since the mapping from a simulated net to its drawing is based solely on the net name, it is possible that transitions and places in a net instance window do not fire in accordance to the visible net structure.

The `de.renew.netPath` property is re-read every time a simulation is started. It is configurable at runtime in the editor's **Configure Simulation** dialog (see Section 4.3.6).

### 2.6.7 Database Backing

You can run Renew using a database that keeps a persistent copy of the current simulation state. In the case of a system crash, this allows you to continue the simulation from a valid state just before the crash when Renew is restarted. Database backing is only supported when the simulation is started from the command line (see Section 4.5 for information on controlling a simulation from command line). Using database backing in gui mode may lead to unexpected behavior. But note that this feature is still experimental and has not been tested across platforms due to lack of funds.

The setup of the persistent database backing is definitely more tricky than the other Renew options, but it is supported by some predefined scripts. The source package contains among others the SQL script `initTable.sql` which can be found in `Simulator/src/main/java/de.renew.simulator/de/renew/database/entitylayer/`.

It creates the required database tables for an Oracle server. For other databases, the script will need some changes, but even the various versions of Oracle differ enough to cause minor problems. A database backend, which supports transactions is required (e.g. MySQL does not support transactions when using the default MyISAM engine; use InnoDB instead).

Having created the tables, you should configure a set of properties to enable the database backing feature (see Table 2.2). These properties are evaluated each time a simulation is set up. The class names specified for the `driver` and `dialect` properties should be accessible

Property name	Type	Comment
<code>de.renew.simdb.driver</code>	class	JDBC driver class (mandatory).
<code>de.renew.simdb.url</code>	URL	JDBC connection URL (mandatory).
<code>de.renew.simdb.dialect</code>	class	Subclass of <code>de.renew.database.entitylayer.SQLDialect</code> (optional).
<code>de.renew.simdb.user</code>	string	User account for database login (optional).
<code>de.renew.simdb.password</code>	string	Password for database login. (ignored when <code>de.renew.simdb.user</code> is not set).

Table 2.2: Properties to configure database backing

via the `de.renew.classPath` (see Section 2.6.5). The `dialect` class is an internal class that adapts Renew to the SQL dialect of your database. The default is the generic `SQLDialect`, but for some databases we already provide experimental implementations like `OracleDialect`, `MySQLDialect` or `MSqlDialect`.

When using the database backing, your nets must conform to certain restrictions. Unfortunately, these restrictions cannot be checked by Renew automatically, so that you must take special care when preparing your net.



All tokens used in your net must be serializable, i.e., custom classes must implement `java.io.Serializable`. Typically, all tokens are also immutable *value objects*, which acquire their state once during creation, before these objects are used as tokens in the net. For value objects, the `equals()` method must not be based on object identity, but on the represented value. Similarly, the `hashCode()` method must also be properly defined.

If you use mutable, stateful objects in your nets, you must observe further restrictions. Contrary to value object, stateful objects must preserve the original implementations of `Object.equals()` and `Object.hashCode()`. Furthermore, the stateful objects must either occur directly as tokens in the net or there must be exactly one token by which a given stateful object is reachable. Failure to do so will result in a corrupted simulation state when restoring the simulation from the database.

There is no garbage collection when using the database-backed simulation.

## 2.6.8 Remote Simulation Access

Any Renew simulation, regardless whether it is started from the command line or within the graphical editor, can be published via Java's Remote Method Invocation (RMI) technique. Any Renew editor can then connect to the published remote simulation, display the token game, and control the firing of transitions. See Section 4.3.6 about how to connect to a running remote simulation. This section focuses on how to configure the simulation engine for remote access.

The first step is that you start an RMI registry on the machine where the server will be running. This is a program distributed together with Java that stores RMI objects and makes them accessible to other VMs. Simply run the program `rmiregistry` (e.g. as a background task). Note that the `rmiregistry` process either needs a classpath with `renew5.0-Pre/plugins/de.renew.remote-2.0.2.jar` included or requires the simulation server to be configured with the `java.rmi.server.codebase` property (see Java RMI docs for details).

Property name	Type	Comment
<code>de.renew.remote.enable</code>	boolean	Enables remote access (defaults to <code>false</code> ).
<code>de.renew.remote.publicName</code>	string	Name to use for RMI registration (defaults to <code>default</code> ).
<code>de.renew.remote.serverClass</code>	class	Implementation of the interface <code>de.renew.remote.Server</code> (defaults to <code>de.renew.remote.ServerImpl</code> ).
<code>de.renew.remote.socketFactory</code>	class	Implementation of the interface <code>java.rmi.server.RMISocketFactory</code> (defaults to RMI's default factory).
<code>de.renew.remote.rmi-host-name</code>	string	the rmi server hostname to the correct ip if the remote server cannot be found. It is similar to the java property <code>java.rmi.server.hostname</code>

Table 2.3: Properties to configure a remote simulation

The remote simulation can be configured by using the properties listed in Table 2.3. The defaults are suitable for most cases, so you just need to enable the remote access by setting the property `de.renew.remote.enable` to `true`.

The `publicName` property is required, if you intent to run several servers on one machine. When connecting from an editor, you can specify the server to connect to by its name.

The `serverClass` and `socketFactory` properties are normally not needed. Plug-in developers may use these properties to replace the default implementations by enhanced versions.



The server class determines the simulation control features which are remotely accessible. The socket factory may enhance RMI transmissions by compression and/or encryption of the network traffic.

All properties except `socketFactory` are re-evaluated at each simulation setup. They are configurable at runtime in the editor's **Configure Simulation** dialog (see Section 4.3.6).

### 2.6.9 Logging

Renew uses the Apache Log4j logging framework to give detailed feedback about its activities (a forked version of Log4j 1.2.17 called Reload4j [15]). The Log4j framework allows users to tailor the logging output to fit their needs. The level of detail can be configured for every application package or class individually.

The Renew base installation contains a binary distribution of the Reload4j framework in the `renew5.0-Pre/libs` directory. The package is loaded immediately at application startup, before any configuration of the plug-in system is done. The early load time of this package has the consequence that the enhanced configuration methods mentioned at the beginning of this Section are *not applicable* to configure the logging system.

#### Configuration mechanism

The configuration options of Log4j would occupy too much space in this manual, please have a look at the documentation section of the Log4j homepage [1]. With the Renew distribution comes a commented default configuration file `renew5.0-Pre/config/log4j.properties`.

Renew looks for Log4j configuration files at similar locations like its own configuration files `.renew.properties` (see Section 2.6.1). The configuration files can be in XML syntax or in the Java properties format and must carry appropriate file extensions. The first file found in the following list of candidates is used:

1. A file named in the system property `log4j.configuration` at the `java` command line with the `-D` option.
2. `.log4j.xml` in the `.renew` subdirectory of your home directory.
3. `log4j.xml` in the installation directory `renew5.0-Pre/config/`.
4. `.log4j.properties` in the `.renew` subdirectory of your home directory and `log4j.properties` in the installation directory `renew5.0-Pre/config/`. If both files exist, individual settings in the user file take precedence over settings in the installation file.
5. A failsafe setup with a simple console logger is used if all configuration files are missing.

Renew provides the `logs.home` property that can be referred from within the configuration files. This property by default points to the directory `.renew/renewlogs` in your home directory, but you can override that setting with the `-D` option on the `java` command line.

#### Default configuration

The default configuration file `renew5.0-Pre/config/log4j.properties` uses the `logs.home` property (see above) so that you get two log files (`renew.log` and `simulation.log`) in that directory. The former comprises application messages while the latter stores simulation traces. In the default configuration, application log messages are also printed to the Java console, but the simulation trace is not. The logging plug-in provides a graphical user interface that displays the simulation trace and allows flexible configuration of logged events (see Section 4.3.6).



## 2.7 Plug-ins

As of Renew 2.0, the application is controlled by a plug-in system. The system is started through the class `de.renew.plugin.Loader`. The loader sets up some class libraries and then loads the main plug-in manager. The plug-in manager finds, loads, initializes and terminates plug-ins, but it knows nothing about Petri nets. The Renew functionality is provided by a set of plug-ins. More plug-ins can be installed to extend Renew.

In this section, there will be a lot of examples with file names and directories. These are all given in Unix syntax. Users of other operating systems: please transform these examples to your appropriate syntax.

### 2.7.1 Install Plug-ins

There are two ways to install a plug-in. If it comes in one single `.jar` file which includes a file named `plugin.cfg`, you can just copy the file in the `renew5.0-Pre/plugins` directory (if you use the MacOS application bundle, this directory is located *inside* the application package at the path `Contents/Java/plugins`). If the plug-in comprises several files, one of which is the file `plugin.cfg`, then you can create a subdirectory below (e.g. `renew5.0-Pre/plugins/myplugin-1.0` and copy all files in this directory.

On the next startup, the plug-in system will find and include the plug-ins automatically. If you want to include the plug-in in the running system, use the `load` command (see Section 2.7.4) in addition.

If the plugin requires additional libraries, install them in the `libs` directory in a subdirectory named after your plugin. For example, for the plugin `renew5.0-Pre/plugins/myplugin-1.0` or `renew5.0-Pre/plugins/myplugin-1.0.jar`, the `libs` are in `renew5.0-Pre/libs/myplugin-1.0`. The `libs` directory has to be installed on the same directory level as the `plugins` directory.

If you do not want to install the plug-in to your `renew` distribution directory, you can install it to some other directory, let's say `/home/myself/plugins/myplugin-1.0.jar` (as single-jar plug-in) or `/home/myself/plugins/myplugin-1.0/` (as multi-file plug-in). Note that the subdirectory has to be named 'plugins'. Then, you can make the plug-in system aware of the plug-in at the next startup by adding the line

```
pluginLocations=/home/myself/plugins
```

to your `.renew.properties` file (see Section 2.6.1 for details). When entering multiple paths, they must be separated by the system's path separator (which is `;` on Windows and `:` on Unix systems). To load the plug-in immediately but temporarily into a running plug-in system, use the `load` command. Depending on the number of files comprising the plug-in, it's one of the following commands:

```
load file:/home/myself/plugins/myplugin.jar
load file:/home/myself/plugins/myplugin/plugin.cfg
```

### 2.7.2 Exclude Plug-ins Temporarily

To hide installed plug-ins from the plug-in finder at startup, you can specify the property `de.renew.plugin.noLoad` either via `-D` command line option or the `.renew.properties` file (see Section 2.6.1 for details). The value of the property is a comma separated list of plug-in names. For example, the line

```
java -Dde.renew.plugin.noLoad="Renew_JHotDraw" \
-p path/to/Renew:path/to/Renew/libs \
-m de.renew.loader.jar gui
```

will start the plug-in system, but terminate with the complaint that the `gui` command is unknown. Because the `JHotDraw` plug-in has not been loaded, all dependent plug-ins can



also not be loaded. This affects the Renew Gui plug-in which would otherwise have provided the `gui` command. Of course it would make more sense to use some non-graphical command like `startsimulation` instead.

Alternatively it is possible to specify only the plug-ins you want to load at startup. For that you have to set the property `de.renew.plugin.autoLoad` to `false` and specify the plug-ins you want to load with the `de.renew.plugin.load` property. The following line will also start the plug-in system and terminate, because of the missing `gui` command.

```
java -Dde.renew.plugin.autoLoad=false \
-Dde.renew.plugin.load="Console, Renew_Simulator, Renew_Formalism, \
Renew_Util" \
-p path/to/Renew:path/to/Renew/libs \
-m de.renew.loader.jar gui
```

### 2.7.3 System Termination

The plug-in system tries to detect the situation where no plug-in is active and therefore the system can be shut down. Plug-ins are “active” if they have some long-term work to do. This always holds for the windows of the graphical editor. A running simulation also counts as active. The Console plug-in has a special keep-alive flag which marks it as active (see Section 2.7.5).

Besides automatic termination, the plug-in system can be terminated by request. The `exit` command (see Section 2.7.4) has just that purpose. The editor may also terminate the plug-in system when it’s main window is closed. The editor does this not by default, you have to set the property `de.renew.gui.shutdownOnClose` to `true`.

The property `de.renew.gui.autostart` automatically starts the editor without the need for an initial `gui` command, when set to `true`. The combination of these two `gui` properties frees users of the pure graphical editor of most complications introduced by the plug-in system.

### 2.7.4 Commands

As mentioned in Section 2.4, the plug-in system needs an initial command to start some plug-in. Any plug-in can provide such commands (although the `gui` command is the one that you will use most of the time). In the following, we present the basic commands provided by the plug-in manager itself and some additional commands provided by other plug-ins. Note that it is also possible to define a chain of commands by separating the commands with `---`. Most of the commands presented here you will typically not use on Renew start-up, but rather in combination with the Console plug-in described in the following Section 2.7.5.

A basic set of commands is provided by the plug-in manager itself:

**help** prints a list of all available commands. Due to the addition or removal of plug-ins, this list may vary from time to time.

**get** prints the value of a property. The property name has to be given as an argument.

`-a` shows all known property keys.

**set** sets the value of a property (as explained in Section 2.6.1). This command accepts multiple arguments of the form `key=value`. It is important that no spaces are included in the key, the value, or in between.

**list** prints a list with the names of all loaded plug-ins. The command respects some mode switches:

`-l` (or `-long`) shows date and version information of plug-ins (if available).



- j (or -jar) shows the jar file locations of plug-ins and libraries.
- o (or -ordered) shows an alphabetically ordered list of plug-in names.

**info** prints information about one plug-in. The plug-in's name has to be specified as command argument (use **list** to see the plug-in names).

**load** loads one plug-in dynamically, if possible. The argument to this command is a URL specifying the plug-in's location. The plug-in location can be given relative to the `de.renew.pluginLocation` directories. For example, the execution of the command `load file:de.renew.gui-2.3.0.jar` would load the gui plug-in from the renew distribution `renew5.0-Pre/plugins/de.renew.gui-2.3.0.jar`. You can use wildcards (e.g. `load *gui*`), if there is an ambiguity the alternatives are prompted for selection of the correct one (only in Console plug-in, not in GuiPrompt).

**unload** terminates and unloads a plug-in, if possible. The plug-in's name has to be specified as command argument (use **list** to see the plug-in names). If other plug-ins depend on the given plug-in, the plug-in system will complain. You can add the argument `-v` to see a list of dependent plug-ins, or the argument `-r` to unload all recursively dependent plug-ins.



Although the plug-in is terminated and all dependencies are cleaned, its classes may still be in memory if there is a reference still held somewhere. Properly cleaning up plugins is ongoing work.

**packageCount** prints the packages and the total number of packages in the class loader.

**exit** terminates the plug-in system, and in consequence the whole application. If some plug-in hangs during termination, you can use **exit force** to kill the Java VM abruptly or **exit ifidle** to exit only if all plug-ins are inactive.

**gc** triggers the Java garbage collector.

**script** loads commands from a text file and executes them. The file name has to be given as argument, it can be specified relative to the current directory. This command is especially useful as initial command, when you in fact need to issue several commands at startup of the plug-in system.

**sleep** This command waits for a given time until the next command is executed. The time to wait is given as argument in seconds.

The following commands are not provided by the plug-in manager, but by some Renew plug-ins. So they are available only when the respective plug-in is loaded.

**gui** starts the graphical editor and/or passes its arguments to the editor. The arguments are supposed to be drawing file names. This command is provided by the Gui plug-in.

**demonstrator** opens a window with a list of drawing file names. This plug-in is provided by the Gui plug-in.

**ex** exports a drawing into various formats. The usage of the command is **ex <type> <drawing>** where **<type>** may be (but may vary depending on the installed plug-ins):

- EPS
- PDF
- PNG
- PNML-P/T-Net
- PNML-P/T-Net-with-Renew-specific-inscriptions



- PNML-Reference-Net
- SVG
- ShadowNetSystem
- Woflan

This command is provided by the Export plug-in which in turn uses the FreeHEP project for graphical exports. The command respects some mode switches:

- a (or -accumulate) n-to-1 export (only available for some formats, e.g. ShadowNetSystem).
- o (or -output) specify the output file.

You can type just **ex** without any options to see the available formats.

**startsimulation** starts a simulation without using the graphical editor. See Section 4.5. This command is provided by the Simulator plug-in.

**simulation** controls a running simulation without using the graphical editor. Use one of the subcommands **help**, **run**, **step**, **stop** or **term** as argument. See Section 4.5. This command is provided by the Simulator plug-in.

**setFormalism** chooses the formalism given as argument as current formalism. This is equivalent to selecting a formalism in the menu (see Section 4.3.6). This command is provided by the Formalism plug-in.

**listFormalisms** lists all installed formalisms. The names listed by this command can be used as argument to the **setFormalism** command or as value for the **renew.compiler** property. This command is provided by the Formalism plug-in.

**keepalive** displays and manipulates the keep-alive feature of the Console plug-in (see Section 2.7.5).

**guiprompt** opens a graphical prompt dialog (see Section 2.7.5), if the Gui Prompt plug-in is installed.

**navigator** opens the Navigator window.

## 2.7.5 Console

The Console plug-in is part of the base distribution. It enhances the plug-in system by an interactive command shell in the Java console window and allows to issue commands to the plug-in system at runtime.

If you start the **renew** from a shell, you will be presented with the prompt **Renew >** at the Java console. Here you can enter any of the commands mentioned in the previous subsection. The Console plug-in itself offers one command with two alternative arguments:

**keepalive on** enables the keep-alive feature. As long as this feature is on, the plug-in system will not terminate automatically because there is no active plug-in (see Section 2.7.3). However, an explicit termination request will still be executed.

**keepalive off** turns the keep-alive feature off.

**keepalive** without argument displays the current state of the keep-alive flag.

The initial state of the keep-alive flag is determined from the value of the property **de.renew.console.keepalive**. If you tend to use the Console plug-in as your only active plug-in most of the time, you should consider adding the line **de.renew.console.keepalive=true** to your **.renew.properties** configuration file.



As an alternative to the Console plug-in, there also exists a gui prompt plug-in for download. This plug-in adds a small dialog to the **Plugins** menu of the editor which accepts commands to the plug-in system. The graphical prompt automatically keeps the plug-in system alive as long as its dialog is open. The dialog can also be opened by the command-line command `guiprompt`. The command feedback is now visible in the dialog instead of the Java console window.

It is possible to escape whitespace characters by surrounding double quotes or with a preceding backslash (e.g. `gui "foo bar/file.rnw"` or `gui foo\ bar/file.rnw`).



The Console-Plugin does not work properly when using the Git Bash on Windows. If it is intended to be used, enter `zZ: -Djline.terminal=jline.UnixTerminal` as a parameter when starting Renew via the Git Bash.

## 2.7.6 Net Components

Net Components (NC) are sets of net elements (Figures) that are grouped in a flat and weak fashion. The aim is to be able to allow to move the whole set of net elements (the net component) in convenient way. *Weak* means that, although the net component can be moved by clicking and dragging the mouse in between the net elements, they can each individually be manipulated. Individual net elements can be dragged resulting in a manipulated layout of the net component. *Flat* means that grouping is not hierarchical. In order to provide the functionality a new Figure has been added, which has no graphical representation of its own.

The Net Component Plugin provides two basic functionalists. The grouping and ungrouping of a selection of net elements and the management of tool palettes, which represent a repository of (pre-) defined sets of net components. A repository consists in a folder that holds several Renew net drawings, an images folder and a configuration file (*.sequence*). The images folder should contain icons in gif format (24×24 pixels), which are used to fill the tool buttons of the palettes. If no image is defined for a specific net component a generic image (*generic.gif*) will be used instead. Names of net drawings and icon images should correspond. The *.sequence* file contains the names of the drawings (without extensions) and define the sequence in which the buttons are shown in the palette. The file may be empty but has to be present in order for the directory to be recognized as net component repository. Further, non-listed drawings are included in the palette without specific order.

There are two ways to use the net component repositories within the plug-in. First, repository folders can be opened directly using the menu command. Second, plug-ins can extend the functionality of the NC plugin by providing a repository. Such a plugin can use the provided generic plugin facade *PalettePlugin* or a customized facade class can be provided. In the latter case also other tool buttons and command line commands can be added to the system. Plugins using the *PalettePlugin* class must provide two properties for the definition of the directory and two for the automatic initialization switch: `de.renew.nc.dir-prop-name`, `de.renew.nc.init-prop-name`. A minimal example for a plugin configuration file (*plugin.cfg*) is presented below. More information about the concept and the tool can be found in [3].

```
mainClass = de.renew.netcomponents.PalettePlugin
name = Renew ExampleComponents
provides = com.example.nc
requires = de.renew.util,de.renew.gui,de.renew.nc
de.renew.nc.dir-prop-name=com.example.nc.dir
de.renew.nc.init-prop-name=com.example.nc.init
com.example.nc.dir=tools
com.example.nc.init=false
```



## 2.8 Troubleshooting

A few possible problems and their solutions are described here. If you have problems, contact us. If you have solutions, contact us, too.

- Java is not found.  
Probably the shell scripts try to look for Java in the wrong places.  
Enlarge the environment space for the command window.
- `java.lang.module.FindException: Module de.renew.loader not found`  
This error occurs if the module path (the list that is set with the `-p` parameter) is not set correctly. Make sure that you are using the correct list separator (`:` for Unix and `;` for Windows). You should verify whether your paths must be set to `Renew:Renew/libs` (base package) or `Renew/dist:Renew/dist/libs` (source package).
- Renew starts, but the window titles are incorrect under the X Windows System.
- I cannot open the sample files.  
Sometimes you need to add the root directory `/` (or `\`, depending on your operating system) to your class path.
- Under MacOS, using `cmd-Q` (`apple-Q`) to quit the application will not give you the opportunity to save changes, even if the document has recently been modified.  
The optional AppleUI plug-in (see Section 2.5.1) solves that problem.

## 2.9 History

**Renew** Since the history of Renew has been more thoroughly documented on <https://www2.informatik.uni-hamburg.de/TGI/renew/history.html>, the actual changelog  
**5.0-Pre** has been removed from the manual.



## Chapter 3

# Reference Nets

First, we are going to take a look at Petri nets with Java as an inscription language. Then we look at synchronous channels and net references, two extensions that greatly add to the expressiveness of Petri nets as described in [9] and [10]. Finally, we are going to see how nets and Java can seamlessly interact with each other. Reference nets and their theoretical foundation as a whole are defined in [11] (in German).

### 3.1 Net Elements

Reference nets consist of *places*, *transitions*, and *arcs*.

There are many types of arcs. Firstly, ordinary *input* or *output arcs* that come with a single arrow head. These behave just like in ordinary Petri nets, removing or depositing tokens at a place. Secondly, there are *reserve arcs*, which are simply a shorthand notation for one input and one output arc. Effectively, these arcs reserve a token during the firing of a transition. Thirdly, there are *test arcs*, which have no arrowheads at all. A single token may be accessed, i.e. tested, by several test arcs at once. This is important, because an extended period of time might be needed before a transition can complete its firing. For a more detailed treatment of test arcs see [6].

Besides these basic arc types, there are arc types that add greatly to the expressiveness of nets, but are not as easy to understand. We postpone the description of these arcs until Section 3.10.

Each place or transition may be assigned a *name*. Currently, this name is used only for the output of trace messages. By default, names are displayed in bold type.

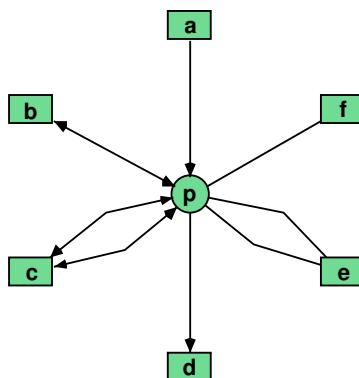


Figure 3.1: The net `elements`



In Fig. 3.1 you can see a net that uses all net elements that were mentioned so far. You can find it in the directory `samples/simple` of the Renew distribution. A single place `p` is surrounded by six transitions. Initially, the place is unmarked. Assume that transition `a` fires, which is always possible, because all its arcs are output arcs. Now one token is placed in `p`, and all transitions except `c` are activated. Transition `c` is still disabled, because it reserves *two* tokens from `p` while it fires. In contrast to this, transition `e` may fire, because it is allowed to test a single token twice. If `a` fires again, transition `c` becomes activated, too, because a second token is now available. A firing of the transitions `b`, `c`, `e`, and `f` does not change the current marking. However, transition `d` will remove one token from `p` during each firing.

Every net element can carry semantic inscriptions. Places can have an optional *place type* and an arbitrary number of *initialization expressions*. The initialization expressions are evaluated and the resulting values serve as initial markings of the places. In an expression, `[]` denotes a simple black token. By default, a place is initially unmarked.

Arcs can have an optional *arc inscription*. When a transition fires, its arc expressions are evaluated and tokens are moved according to the result.

Transitions can be equipped with a variety of inscriptions. *Expression inscriptions* are ordinary expression that are evaluated while the net simulator searches for a binding of the transition. The result of this evaluation is discarded, but in such expressions you can use the equality operator `=` to influence the binding of variables that are used elsewhere.

*Guard inscriptions* are expressions that are prefixed with the reserved word `guard`. A transition may only fire if all of its guard inscriptions evaluate to `true`.

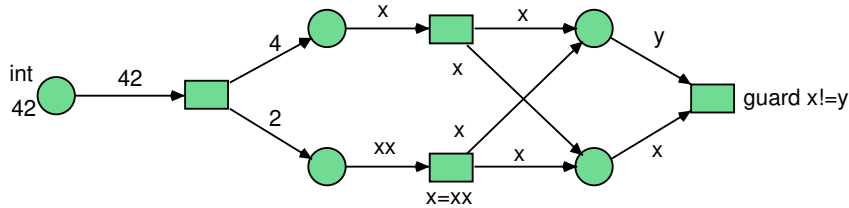


Figure 3.2: The net colored

With these additions we cover the basic colored Petri net formalism. In Fig. 3.2, which is also provided in the directory `samples/simple`, we find a net that uses the basic place and arc inscriptions. At the left, we have a place that is typed `int`, which means that it can only take integers as tokens. In this case, it has an initial marking of one integer 42 token. The other places are untyped and initially unmarked. The leftmost transition will take 42 out of the place and deposit one 4 and one 2 into the respective places. The upper middle transition takes some `x`, which happens to be 4 in this case, out of its input places and copies it into its two output places. The lower middle transition is similar, but here the equality of input and output arc variables is established by the transition inscription `x=xx`. The rightmost transition has a guard that ensures that `x ≠ y`, written `guard x!=y`. Therefore it can only take a 2 out of the upper place and a 4 out of the lower place or vice versa.

*Action inscriptions* are expression inscriptions preceded with the keyword `action`. Contrary to expression inscriptions, action inscriptions are guaranteed to be evaluated exactly once during the firing of a transition. Action inscriptions cannot be used to calculate the bindings of variables that are used on input arcs, because input arc expressions must be fully evaluated before a transition can fire. However, action inscriptions can help to calculate output tokens and they are required for expressions with side effects.

Then there are *creation inscriptions* that deal with the creation of net instances (see Section 3.6) and *synchronous channels* (see Section 3.7). But first we will look closer at the expression syntax, which is very similar to a subset of Java. In fact, we have to look carefully to spot the differences.



<code>boolean</code>	boolean values ( <code>true</code> , <code>false</code> )
<code>byte</code>	8-bit signed integers
<code>short</code>	16-bit signed integers
<code>int</code>	32-bit signed integers
<code>long</code>	64-bit signed integers
<code>char</code>	16-bit unsigned Unicode characters
<code>float</code>	32-bit IEEE floating point numbers
<code>double</code>	64-bit IEEE floating point numbers

Table 3.1: The primitive data types of Java

## 3.2 I do not Want to Learn Java

Even if you do not want to learn Java, Renew might be a useful tool for you, although it loses some of its expressiveness. In many cases it is enough to learn how to write numbers, strings, variables, and the simplest operators.

Reference nets provide extensions that go well beyond simple high-level Petri nets with Java inscriptions. After you have read the next sections, you can use these extensions to generate complex models without the need to incorporate Java code.

But remember that there are always subproblems that are easier to express in a programming language rather than Petri nets. Reference nets work together seamlessly with Java programs and gain a lot from utilizing the Java libraries. So once you *do* learn Java, you can choose the appropriate modeling method for each task at hand.

## 3.3 A Thimble of Java

If you are already familiar with Java, you will want to skip to Section 3.4 where we discuss the differences between Java and the inscription language used in reference nets. Here we give a most rudimentary introduction to Java.

Java is an *object-oriented* programming language, but not everything is an object in Java. There are eight non-object data types in Java which are listed in Table 3.1. The types `byte`, `short`, `char`, `int`, and `long` are called integral types here. Together with `float` and `double` they form the number types.

In Figure 3.3 you can see two type hierarchies. On the left the ordinary Java subtype relation is depicted. You can see that `long` is a subtype of `float` although some loss of precision might occur during the conversion. Nevertheless, Java will silently insert this conversion whenever it is required in a program.

Although this is helpful for Java programs, it poses several problems in the context of Petri nets, where the direction of information transfer is not always immediately obvious. Hence such conversions are not done by the simulator. Instead we introduced the relation of lossless conversions, which you can find on the right hand side of Figure 3.3. This relation governs the type constraints between places and their neighboring arcs.

All other types except primitive types are *reference types*, i.e., references to some object. Every object belongs to a *class*. When a class is declared, it may receive an arbitrary number of field declarations and method declarations. *Fields* are variables that exist once per object or once per class. The binding of the fields of an object captures the state of that object. *Methods* describe the possible actions of an object. Each method has a name, a list of parameters, and a body, i.e. a sequence of statements that are executed if the method is invoked.

Method declarations and field declarations are nested in the declaration of the class to which they belong. It is possible to use the predefined classes without writing new ones, when working with Renew. We are going to see later how nets themselves can be regarded



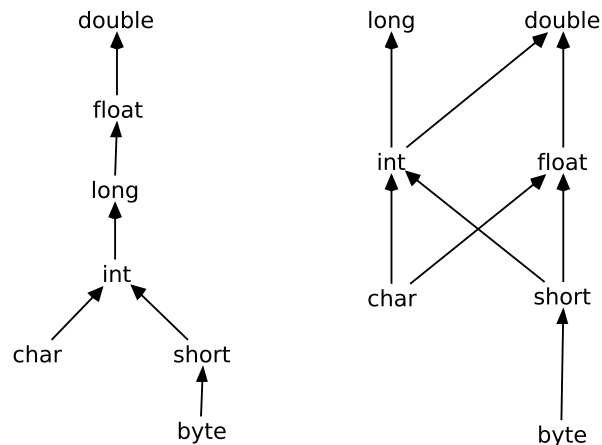


Figure 3.3: The Java type hierarchy and the hierarchy of lossless conversions

as classes. For a detailed discussion of the Java type system and the Java system libraries we refer the reader to the literature.

Now we are going to look at the syntax of Java expressions. We only deal with the subset of Java that is relevant to reference nets.

*Variables* are represented by identifiers. Identifiers are alphanumeric strings starting with a non-numeral character. E.g., `renew`, `WRZLGRMF`, `go4it`, and `aLongVariableName` are all valid variable names. By convention, variable names should start with a lower case character. The declaration of a variable is denoted by prefixing the variable name with the type name, e.g. `int i`. Variables were already silently assumed in Fig. 3.2.

The Java language provides *literals* for integers (123), long integers (123L), floats (12.3F), and doubles (12.3). Furthermore, there are the boolean literals `true` and `false`, string literals (`"string"`), and character literals (`'c'`). Java uses 16-bit Unicode characters and strings. There are no literals for the primitive types `byte` and `short`.

There is also one literal of reference type named `null`. Every variable of a reference type may take `null` as a value. `null` equals only itself and no other reference. Trying to invoke a method of the `null` reference will fail with a runtime exception.

A sizable set of *operators* is provided in Java. Here we are going to discuss those operators that are still present in reference nets. The binary operators are listed in Table 3.2, where we also note their interpretation and the operand types to which each operator is applicable.

Most of the operators are defined for primitive types only, but you can also check if two references are identical with `==` and `!=`.



Never use `==` or `!=` to compare the equality of strings, like in `s1==s2`. Always use the Java-method `equals(...)` as in `s1.equals(s2)` or you will get strange results. This is a peculiarity that annoys many Java beginners, but we are not in a position to change this behavior.

The operator `+` is also used to concatenate strings. If only one operand of `+` is a string, the other operand is first converted to a string and the two strings are concatenated afterward, e.g. `"7x8="+42` results in the string `"7x8=42"`.

If multiple operators are present, they are grouped according to their *precedence*. `*`, `/`, and `%` have the highest precedence, `|` has the lowest precedence. The expression `a+b%c*d|e` is equivalent to the fully parenthesized expression `(a+((b%c)*d))|e`. The order of precedence for each operator can be found in Tab. 3.2. If in doubt, make the parentheses explicit.

An operand of a small type (`byte`, `short`, or `char`) is automatically converted to `int` before any operator is applied. If you need the result as a small type, you have to make an



*	multiply	number
/	divide	number
%	modulo	number
+	plus	number, <b>String</b>
-	minus	number
«	shift left	integral
»	shift right	integral
»>	signed shift right	integral
<	less than	number
>	greater than	number
<=	less than or equal	number
>=	greater than or equal	number
==	equal	primitive, reference
!=	unequal	primitive, reference
&	and	primitive
^	exclusive or	primitive
	or	primitive

Table 3.2: Java binary operators, rules separate operators of equal precedence

-	negate	number
~	bit complement	integral
!	not	boolean

Table 3.3: Java unary operators

explicit *cast*. E.g., `(byte)b1+b2` adds the two bytes `b1` and `b2` and truncates the result to 8 bits. You might also want to reduce the precision of a floating point number by saying `(float)d1` where `d1` is a `double` variable. The opposite case where precision is added, e.g. `(long)b1`, is helpful, too, but usually this kind of conversion is added automatically in the places where it is needed.

Casts between reference types are also possible, but here no conversion takes place. Instead, it is checked that the operand is indeed of the given reference type, either at compile time or at run time, if required. E.g., if a variable `o` of type `Object` is declared, we can say `(String)o` to ensure that `o` does indeed hold an object of type `String`.

There are a few unary operators, too. They are listed in Table 3.3. Unary operators and casts have a higher operator precedence than any binary operator.

A last operator that must be mentioned is `instanceof`. Its left operand is an expression as usual, but its right operand must be the name of a class or interface. It evaluates to true, if the result of the expression is a reference to an object of the given class or one of its subclasses or of a class that implements the given interface.

With an object reference you can also inspect fields and invoke methods. E.g., if there is an object `o` with a field `f`, you can access the field by writing `o.f` inside a Java expression. The result will be the current value of that field.

For an object `o`, a call of the method `m` with the parameters `1` and `x` would look like `o.m(1,x)`. This has the result of binding the formal variables to the parameter values and executing the body statements of the method. Unless the method is of the return type `void`, a return value will be calculated and returned.

Due to *overloading*, there might be more than one method of a given name within some class. In that case, the method that matches the parameter types most closely is invoked.

In order to create a new instance of a class, you can use the `new` operator. E.g., the expression `new java.lang.StringBuffer()` will create a new object that belongs to the



class `java.lang.StringBuffer` and invoke its *constructor*. A constructor can be seen as a special method that initializes a new object. The `new` operator can take arguments inside the parentheses. The arguments are then passed to the constructor just as in an ordinary method call.

## 3.4 The Inscription Language

Because we are dealing with a colored Petri net formalism, the net simulator must determine which kind of token is moved for each arc.

The possible kinds of tokens are Java values or references. By default, an arc will transport a black token, denoted by `[]`. But if you add an *arc inscription* to an arc, that inscription will be evaluated and the result will determine which kind of token is moved.

### 3.4.1 Expressions and Variables

Arc inscriptions are simply Java expressions, but there are a few differences. The first difference concerns the operators that are used in expressions. In Java the binary operators `&&` (logical and) and `||` (logical or) are short-circuit operators. I.e., if the result of the left operand determines the result of the operator, the right operand is not even evaluated. This would imply an order of execution, which we tried to avoid in our net formalism. Hence, the two operators are not implemented. The same holds for the ternary selection operator `?:`. An additional benefit of its exclusion from the language is that this frees up the colon for other syntactic constructs. Possibly, these three operators might still occur in later releases of Renew.

In Java variables receive their value by *assignment*. After a second assignment, the value from the first assignment is lost. This flavor of variables is not well-suited for high-level Petri nets. Instead variables are supposed to be bound to one single value during the firing of a transition and that value must not change. However, during the next firing of the same transition, the variables may be bound to completely different values. This is quite similar to the way variables are used in logical programming, e.g. in Prolog.

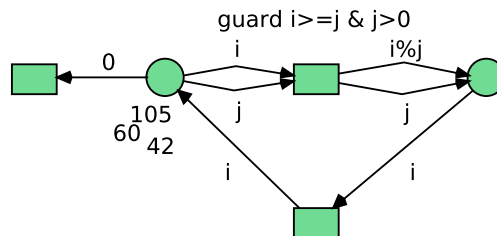


Figure 3.4: The net `gcd`

In Fig. 3.4 we show an example net that uses expressions as arc inscriptions and also as guard inscriptions. The example is provided in the directory `samples/simple`. Some numbers are put into a place and the net will compute the greatest common divisor of all these numbers and terminate with no more enabled transitions. The upper central transition is the most interesting. It removes two tokens from the pool of numbers, but a guard makes sure that the two numbers are greater than zero and correctly ordered. The transition outputs the smaller number and the remainder (denoted by the operator `%`) of the division of the greater number by the smaller number. The lower central transition simply puts the new numbers back into the pool and the left transition discards zeroes.

Note how a single variable can be bound to different values at different times. Note that the simulator will automatically search for possible bindings of the variables.



### 3.4.2 Types

For reference nets, types play two roles. A type may be an inscription of a place. This means that the place can hold only values of that type. The net simulator can statically detect many situations where type errors might occur, i.e., when transitions try to deposit tokens of the wrong type into a place. Furthermore, variables may be typed. This means that the variable can only be bound to values of that type.

In Java every variable needs to be declared. There are of course many good reasons to demand this, but there are times when it is valuable to write programs without having to worry about a type declaration. One of these cases are throw-away prototypes, which are supposed to be developed very quickly. Petri nets are generally usable for prototyping, so we wanted to be able to write nets without having to declare variables.

But for stable code that will be used in a production application types are a must. Therefore reference nets provide the option to create a *declaration node*. In the declaration node, an arbitrary number of Java import statements and Java variable declarations are allowed. If a declaration node is present, then *all* variables must be declared. This means that you have the choice between complete liberty (no variables can be declared) and complete security (all variables must be declared).

Note that an undeclared variable does not have a type. Therefore, the type of an expression can only be determined at runtime, if it contains undeclared variables. Worse, if a method is overloaded, the choice of the actual method must be delayed until runtime when all operator types are known. This is contrary to ordinary Java, where overloaded methods are disambiguated at compile time.

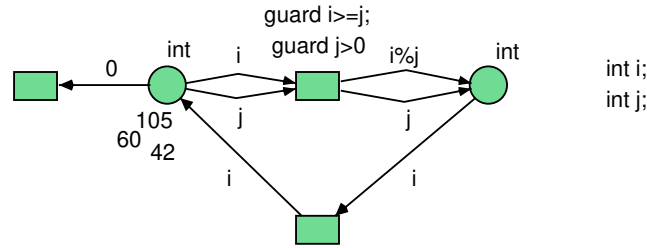


Figure 3.5: The net gcdtyped

Fig. 3.5 shows a typed variation of the greatest common divisor algorithm. First, you can see the type inscriptions of the places that are all `int` in this case. Second, you will notice the declaration node where the two variables are declared. As in Java, declarations consist of the type followed by the name of the variable.

Places can be typed, too. This allows the simulator to catch some difficult situations before the actual simulation. For input arcs, the type of the arc inscription should be comparable to the type of the place, i.e. either a subtype or a supertype. Otherwise it is probable that the expression yields a value that cannot be a token in the place. Note that for this type check we have to use the lossless conversion rules as depicted in Figure 3.3

For output arcs we require that the type of the arc expression is narrower than the type of the place, so that the place can always take the resulting token. This is important, because the values of the output expressions might only be determined during the firing of the transition when it is too late to declare the transition disabled. For input arcs we can simply ignore any binding that would result in a token of a bad type.

As a special case it is required that an output arc expression for a typed place must be typed. In practice this means that you have to declare your variables as soon as you assign types to places. On the other hand, you can type the variables without having to type the places.



Sometimes it is required to convert an expression of one type to an expression of a different type. Reference nets support Java's concept of *casts*. A cast is indicated by prefixing an expression with the desired type enclosed in parentheses. E.g., `(Object)"string"` would be an expression of type `Object`, even though it will always result in `"string"`, which is of type `String`.

On the other hand, if you know that a variable `o` of type `Object` will always hold a string, you can say `(String)o` to inform the type system of this fact. For primitive types, a conversion takes place, e.g., `(byte)257` converts the 32-bit integer 257 into the 8-bit integer 1 by truncating the most-significant bits.

In Fig. 3.5 we also illustrated that you can make multiple inscriptions to a single transition, as we have two guards for a single transition.

If multiple transition inscriptions are given in a single graphical figure as in this case, the inscriptions have to be separated by semicolons. They may also optionally be terminated with a semicolon.

### 3.4.3 The Equality Operator

If we look at the new semantics of variables, we might wonder what the meaning of the operator `=` is. It cannot be an assignment, because variables are immutable. Instead, it is merely a specification of equality. You will usually want equality specifications to occur inside special inscriptions that are attached to transitions. E.g., you can say `x=2` to bind the variable `x` to 2 or you could use `x=y*z+42` for a more interesting computation. If you specify both `x=2` and `x=3` for a single transition, that transition will not be able to fire, because `x` cannot be bound in a way that matches both specifications.

Keep in mind that `=` is based on equality in the sense of the `equals(Object)` method and not in the sense of the operator `==`. This might confuse experienced Java programmers, but it is the only possibility to avoid certain other anomalies.

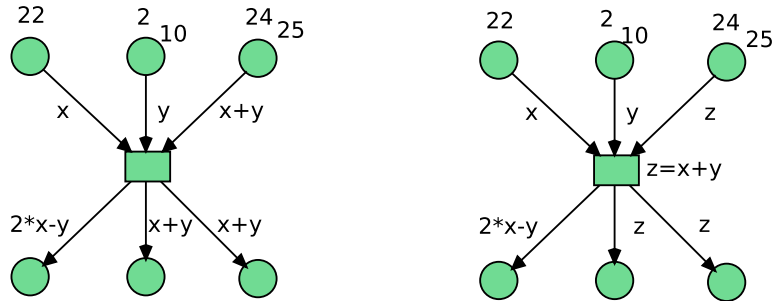


Figure 3.6: The net `equality`

In the net from Fig. 3.6 you can see two transitions that perform equivalent actions, as you can see when you load the nets from `samples/simple` and simulate them. The transition on the right uses a variable `z` to hold the value of the computation `x+y`. At the left we see an example where an expression occurs on an input arc. Such expressions are properly evaluated and the simulator checks whether the resulting token is available.

But expressions on input arcs have to be used with care. Just because the simulator knows that `x+y` equals 24 and `x` equals 22, it cannot conclude that `y` is 2. Such computations would have been possible in some cases, but not in others. Due to consistency we decided on the general rule that expressions are not evaluated backward. The only exception are type casts, which we met earlier on. A type cast that may be performed without losing information, e.g. `(long)i` for an integer `i`, can be calculated backward. If due to an equality specification the



result of such a cast is known, it is propagated backward to the casted expression, possibly after some conversion.

If a backward computation is desired in the other cases, it has to be made explicit. In our example, we could complement the equation  $z=x+y$  by  $x=z-y$  and  $y=z-x$ . Now the simulator can determine  $y$  from  $x$  and  $z$ . This is allowed, exactly because  $=$  does not mean an assignment but an equality specification. If a bound variable is calculated again by a redundant equation, this does not pose a problem as long as the two bindings are equal.

If  $=$  does not assign, what do the modifying operators  $+=$ ,  $*=$ , and so on mean in reference nets? Simple answer: They make no sense and were therefore excluded from the language. Similarly, the operators  $++$  and  $--$  do not appear.

### 3.4.4 Method Invocations

Reference nets also support method invocations. E.g., `x.meth("a")` invokes the method `meth` of the object referenced by `x` with the parameter string `"a"`. All Java methods can be used in reference nets, but there are some critical points.

First of all, methods can be evaluated more than once. Worse, a method might be invoked even though the transition does not fire. This is done, because the result of a method invocation might be needed to determine whether a transition is enabled at all. Therefore it is best, if the invoked methods do not produce any side effects. If side effects are required, then they should be invoked in action inscriptions only.

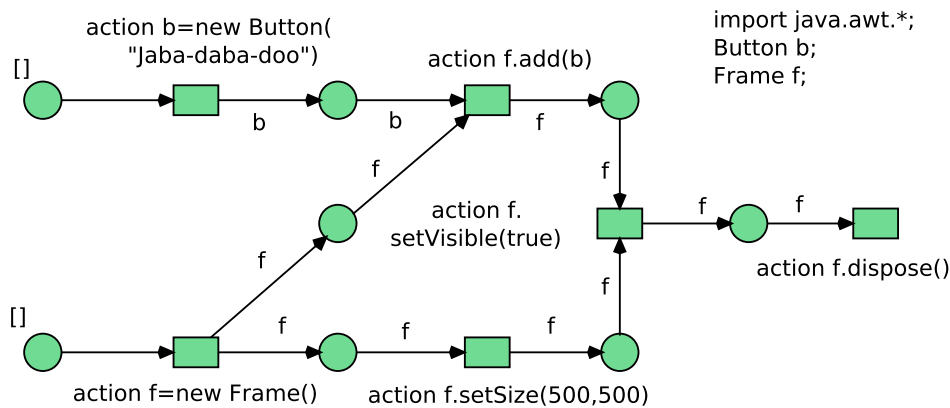


Figure 3.7: The net `frame`

Fig. 3.7 shows some example method calls that are invoked by net inscriptions. The net is saved in the directory `samples/simple`. The declaration node contains an *import statement* that instructs the simulator to search the package `java.awt` for classes whose names appear in the net. The variables `f` and `b` are then declared as a `Frame` and a `Button`. These two classes are in the package `java.awt`, so we could have written `java.awt.Frame` and `java.awt.Button` instead. The procedure that has been implemented here is simple. A window and a button are created, the window is resized and the button is added to the window. Now we can show the window, let the user click it some times, and remove it from the screen again.

It is possible to give multiple actions in a single transition inscription in a semicolon separated list, e.g., `action y=o.m(x); action x=o.m();` would be allowed. Note that the order of execution need not match the textual order. In the previous example, `action x=o.m()` would have to be executed first, because it is required to determine the binding for `x`. In the same sense, the `action` keyword only applies to a single expression, not to



all following expressions. E.g., `action y=o.m(x); x=o.m();` would mean that `x=o.m()` is evaluated early during the search for a binding, because it is not an action.

### 3.5 Tuples, Lists, and Unification

The inscription language of reference nets has been extended to include *tuples*. A tuple is denoted by a comma-separated list of expressions that is enclosed in square brackets. E.g., `[1,"abc",1.0]` denotes a 3-tuple which has as its components the integer 1, the string "abc", and the double precision float 1.0. Tuples are useful for storing a whole group of related values inside a single token and hence in a single place.

In ordinary Java, there are no tuples. If we want to store a group of values, we can simply create a group of variables, each of which holds one value. But with Petri nets we want to store arbitrarily many tokens in a place, making this solution useless in many cases.

It would of course be possible to create a Java class with an appropriate set of fields to wrap a group of values, but this would result in an excessive amount of trivial functionless classes. (By the way, this is what has to be done in Java in some cases, too.)

Tuples are weakly typed. They are of type `de.renew.unify.Tuple`, but their components are untyped. It is not even specified whether a component of a tuple holds a primitive or a reference type.

This does not matter much, because the only operation on tuples (or rather the only operation that should be used) is *unification*. You can unify tuples through an equality specification. E.g., `[x,y,z]=t` means that `t` must be a 3-tuple. Furthermore, `x` will be equal to the first component of `t`, `y` to the second, and `z` to the third.

We already know that the black token is denoted by `[]`. Therefore a black token is simply a tuple without components (a zero-tuple).

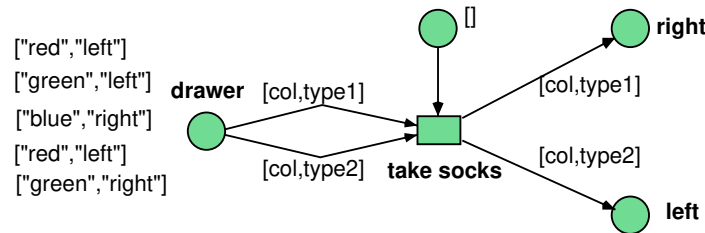


Figure 3.8: The net `socks`

In Fig. 3.8 we can see the sock algorithm of the typical theoretical computer scientist. The scientist will reach into the drawer to fetch two socks. It does not matter if the socks are left socks or right socks (they are topologically equivalent) as long as they are of the same color. In the net, which can be found in the directory `samples/tuple`, this is achieved by using the variable `col` in both arc inscriptions that will remove tokens from the `drawer` place.

Tuples may be nested. `[[1,2],[3,4,5]]` would be a 2-tuple that has a 2-tuple as its first component and a 3-tuple as its second component. This might be useful if the components are hierarchically structured.

It is a common task to use tuples to simulate a database, so that the number of tuples in a place can be considerable. Often one component of an input arc tuple can be determined without accessing the place. In this case, Renew accesses only those tokens that match the known component. Because few tokens need to be checked, the simulation can proceed quickly. If *two* components of the input arc tuple are known, the simulation engine will use that component as a key that results in fewer matches.

In functional programming nested pairs are used as a representation of lists. This could be simulated by nested tuples, but it would result in nets that are hard to read. Hence



we added explicit list support to the language. Lists are delimited by curly braces, e.g.,  $\{1,2,3,4\}$  would be a four element list. Lists, like tuples, support pattern matching. Using  $\{1,2,3,4\}=\{u,v,w,x\}$  as a transition inscription, we would get  $u=1$ ,  $v=2$ , and so on.

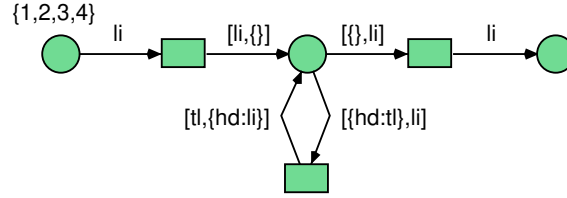


Figure 3.9: The net **reverse**

In order to handle lists of unknown length, a tail expression may be added to the list. The tail expression is separated from the ordinary list elements by a colon. The tail expression matches an arbitrary list of elements. By requiring  $\{1,2,3,4\}=\{u,v:w\}$  we get  $u=1$ ,  $v=2$ , and  $w=\{3,4\}$ . The tail consists of all elements that are not explicitly represented. The tail may be empty as in  $\{1,2,3,4\}=\{u,v,w,x:y\}$  where  $y=\{\}$ . Note that the empty tuple  $[]$  and the empty list  $\{\}$  are not equal.

In Fig. 3.9 you can see an example net, which reverses a list by successively splitting off the head of the original list and appending it to a result list. The remainder of the original list and the result list are jointly contained in a tuple. Once the original list is fully consumed, the result list is extracted.

## 3.6 Net Instances and Net References

When a simulation run of a net is started, the simulator creates a *net instance* of the net that is simulated. A net that is drawn in the editor is a static structure. However, an instance of the net has a marking that can change over time. Whenever a simulation is started, a new instance is created.

Most net formalisms stop here. They create one instance of a net and simulate it. *Renew* allows you to create many instances of a single net. Each instance comes with its own marking and can fire independently of other instances.

Every net has a *name*. The name is derived from the file where it is saved by removing the directory name and the suffix. E.g., a net saved in `/users/foo/bar/baz.rnw` would have **baz** as its name.

New net instances are created by transitions that carry *creation inscriptions*, which consist of a variable name, a colon (:), the reserved word **new**, and the name of the net. E.g., **x:new baz** makes sure that **x** is bound to a fresh instance of the net **baz**.

In Figs. 3.10 and 3.11 you can see a simple example. These nets are available in the `samples/creation` directory.

When you start a simulation of **creator**, the top transition can fire and creates two new net instances of **othernet**. References to the two nets are deposited in the middle places. Now *three* transition instances are activated, namely the two transitions in the two instances of **othernet** and the bottom transition of **creator**. The guard is satisfied, because two different creation inscriptions are guaranteed to create different net instances. You never create the same instance twice.

Now the order of execution is undefined. It might be possible that the bottom transition of **creator** fires first. Even in that case, the two transitions instances of **othernet** remain activated. A net does not disappear simply because it is no longer referenced.

On the other hand, if a net instance is no longer referenced and none of its transition instances can possibly become enabled, then it is subject to garbage collection. Its presence



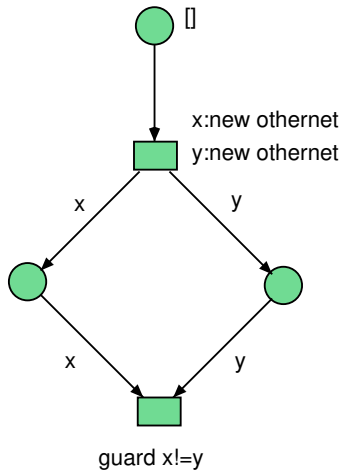


Figure 3.10: The net `creator`

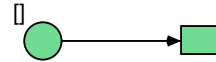


Figure 3.11: The net `othernet`

has become undetectable and hence we might remove it without further ado.

In Java the reserved word `this` denotes the object whose method is currently executed. In reference nets `this` denotes the net instance in which a transition fires.

We are often going to treat net instances like objects of an object-oriented programming language. They are instances of a net, just like objects are instances of a class. They have an identity that can be checked with `==` and `!=` just like objects. They have a state that can change over time and here places seem to correspond to attributes. Net instances also encapsulate data. They can be referenced from other net instances. The only missing component for full objects are methods. In the next section we will learn about a communication concept that can be substituted for method calls sometimes. In Sec. 3.9 we will finally see how nets can be equipped with methods.

### 3.7 Synchronous Channels

Currently, the idea of net instances might not seem interesting, because there is no mechanism by which nets can influence each other. Hence, although net instances encapsulate data, they encapsulate it so well that it cannot be accessed at all.

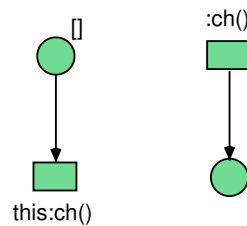


Figure 3.12: The net `synchro`

In this section we will establish a means of communication for net instances. There are two fundamentally different ways of communication. First, we have message passing where a sender creates a message that can be read by a receiver later on. The sender can always send the message regardless of the state of the receiver. The receiver may or may not be



able to process the message. Second, we have synchronous communication where sender and receiver have to agree on participating in an communication at some point of time.

In Petri net formalisms, the former kind of communication is usually captured by so-called fusion places. Reference nets, though, implement the latter kind of communication in the form of *synchronous channels*. This allows more expressive models compared to message passing, because it hides much of the inherent complexity of synchronization from the developer. Furthermore, message passing can always be simulated using synchronous communication.

Synchronous channels were first considered for colored Petri nets by Christensen and Damgaard Hansen in [5]. They synchronize two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters, before they can engage in the synchronization.

Here we generalize this concept by allowing transitions in different net instances to synchronize. In association with classical object-oriented languages we require that the initiator of a synchronization knows the other net instance.

The initiating transition must have a special inscription, the so-called *downlink*. A downlink makes a request at a designated subordinate net. A downlink consists of an expression that must evaluate to a net reference, a colon (:), the name of the channel, an opening parenthesis, an optional comma-separated list of arguments, and a closing parenthesis. E.g., `net:ch(1,2,3)` tries to synchronize with another transition in the net denoted by the variable `net`, the channel has the name `ch` and is passed the parameters 1, 2, and 3.

On the other side, the transition must be inscribed with a so-called *uplink*. An uplink serves requests for everyone. A transition that is called through an uplink need not know the identity of the initiator, just like an activated method of an object does not necessarily know of its caller. Therefore the expression that designates the other net instance is missing for uplinks. An example uplink inscription would look like `:ch(x,y,z)`, which means that the channel name is `ch` and that the three channel parameters must match the binding of the variables `x`, `y`, and `z`.

The uplinks and downlinks of a transition may be given as individual transition inscriptions or in a single semicolon separated list in one inscription. The list might even include action inscriptions and creation inscriptions simultaneously with the channel invocations.

Let us first look at the special case where two net instances within the same net synchronize. This is done by providing the keyword `this` as the target of the downlink. In Fig. 3.12 you can see an example net with local channels, which is provided in the directory `samples/channel` like all other nets of this section. The input place of the left transition is marked and the transition's downlink specification can be met by synchronizing with the right transition. Both transitions fire synchronously, such that one token is removed from the left place and one token is added to the right place in a single step. Now no more transitions are enabled. The left transition lacks a token on its input place, the right transition has an uplink that is not invoked by another transition.

Generally, transitions with an uplink cannot fire without being requested explicitly by another transition with a matching downlink. We will sometimes call a transition without an uplink a *spontaneous* transition. But even a spontaneous transition must find an appropriate synchronization partner if it has a downlink.

It is allowed that a transition has multiple downlinks. It is also allowed that a transition has both an uplink and downlinks. This is exemplified in Fig. 3.13. Again the transition on the left initiates the synchronization. The required channel is offered by the middle transition which does nothing except linking to the channel `bar` twice. This is allowed, a transition may fire multiple times in one synchronous step, although it might be confusing and should be avoided when possible.

In general, multiple levels of synchronization are suspect from a methodical point of view, because they tend to be difficult to understand. Petri nets excel at displaying control flow and it seems that synchronous channels should not be used to encapsulate complex control flows or even loops. It is best to use channels where they show their greatest potential, namely synchronization, communication, and atomic modifications.



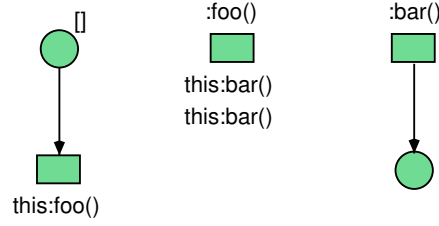


Figure 3.13: The net `multi`

Channels can also take a list of parameters. Although there is a direction of invocation, this direction need not coincide with the direction of information transfer. Indeed it is possible that a single synchronization transfers information in both directions. Fig. 3.14 shows a possible application where the left transition consults a lookup table that is managed by the right net instance. The parameter lists  $(x,y)$  and  $(a,b)$  match if  $x=a$  and  $y=b$ . After binding  $x$  from the left place the variable  $a$  is determined and only one token of the right place matches the tuple of the arc inscription. This allows to bind  $b$  and hence  $y$ .



Figure 3.14: The net `param`

In the previous examples we only encountered local synchronizations within one net, but Figs. 3.15 and 3.16 show two separate nets that can communicate. The net represents the basic schedule of Santa Claus on the night before Christmas. He wakes up, takes a new bag from the shelf and fills it with presents. Later on he can simply reach into his bag and get something that he can put into the children's boots, maybe some candy or a brand new game.

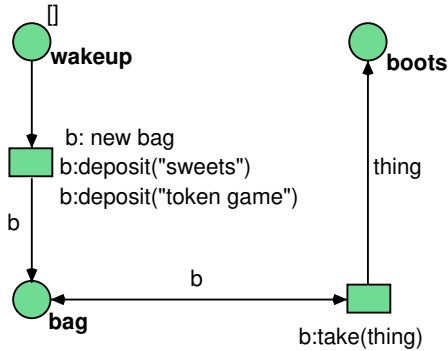


Figure 3.15: The net `santa`

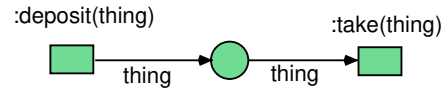


Figure 3.16: The net `bag`

It is possible to create synchronization loops where the invocation of a channel results in the invocation of the same channel. This should be avoided, because it might throw the simulator into an infinite loop. A different search strategy could have avoided this problem, but it would have incurred a significant performance cost.

We mentioned that message passing can be simulated by synchronous channels. The canonical way would be to create a transition with a single-parameter uplink and a single



output arc in the receiving net, which can then put the argument of its uplink into its output place. Because this transition is always enabled, messages can always be sent and the state of the receiver does not influence the sender in any way. After the message has been put into the place, it can be processed in an arbitrary way.

Using the special inscription `x:new net()` you can indicate that you want to create a new net with name `net`, assign it to the variable `x` and perform a synchronization with the uplink `:new()` in the newly created net in one step. Note that, unlike earlier versions of Renew, the channel `new` is only invoked when you request it explicitly. An implicit invocation, even for the initially created net instance, is not performed.

For additional examples, see the nets that are distributed with Renew in the directory `samples`. Not all of them are given a detailed discussion in this manual. In directory `samples/fireman` you can find the fireman example that is based on an idea of Petri [14]. A workflow system of a law enforcement agency is the basis for the nets in `samples/prosecute`. They are based on the article [18], where this example is attributed to W.M.P. van der Aalst.

### 3.8 Manual Transitions

The `manual` inscription, if attached to a transition, stops the simulator from firing the transition automatically. You have to fire such transitions with a right mouse button click. This is useful for simulating external events or for adding control switches.

In the directory `sample/simple` you can find the net `mutex`, which is also displayed in Fig. 3.17. It shows a mutual exclusion algorithm that uses only a single channel per direction for both token transfers and token requests.

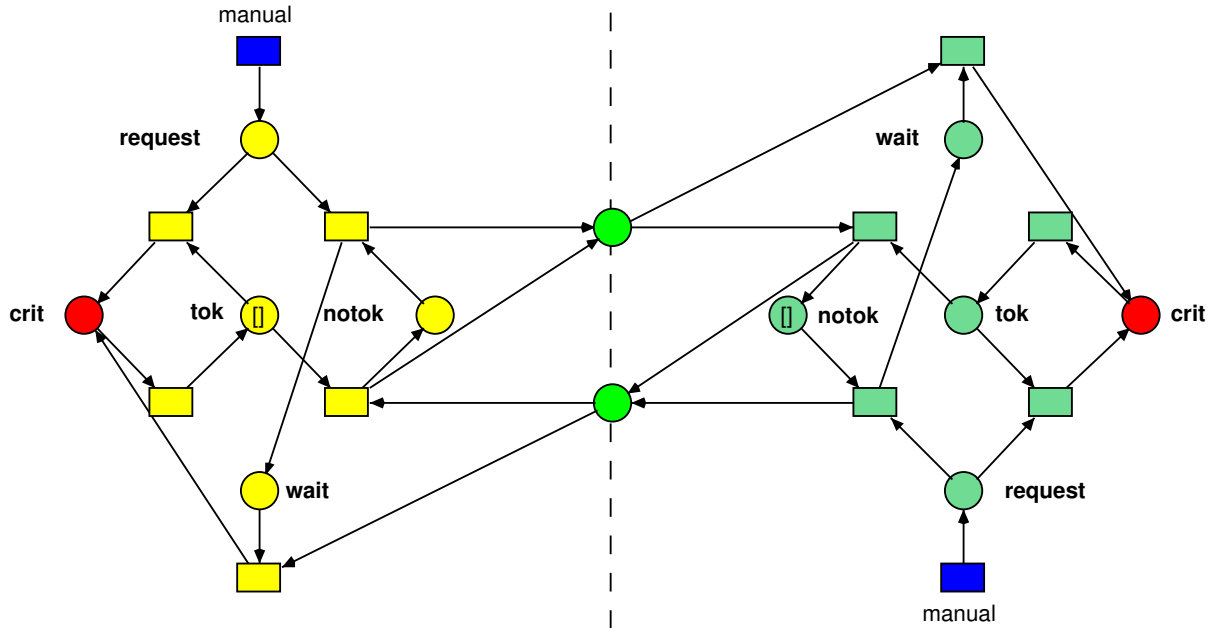


Figure 3.17: The net `mutex`

The critical sections that are guarded by the mutex algorithm are painted red. The left process is shown in yellow, whereas the right process is shown in dark green. Initially, the left process holds the token to enter the critical section in its place `tok`. The right process does not own the token.

After one of the blue `manual` transitions is fired, the associated process tries to gain the



token and enter the critical section. To see this, run the simulation continuously. The `manual` transitions do not fire automatically, but after one of them is fired by using the middle or right mouse button, the rest of the net processes the request immediately.

## 3.9 Calling Nets from Java

In the previous section we considered the use of a Java-like inscription language in reference nets. Now we are going to allow access to reference nets from Java code. Nets are already objects and they have an identity. But up to now all nets have the same type, namely `de.renew.net.NetInstance`, and implement the methods of `de.renew.net.NetInstance` only.

### 3.9.1 Net Methods

Therefore we must create new classes that behave like nets when treated by the simulator, but which implement additional methods. Upon invocation, the methods can communicate with the net through synchronous channels, which will in turn take the required actions. These classes will be known as *stub classes*.

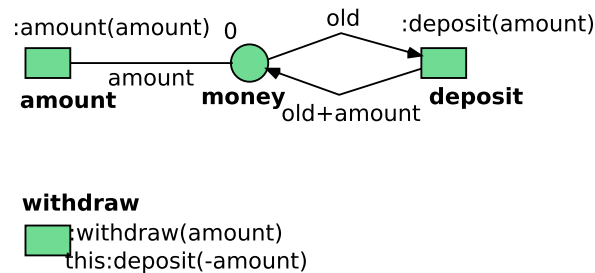


Figure 3.18: The net `account`

The net from Fig. 3.18 models a very simple bank account. The customer can only deposit and withdraw money and view the current amount. But we still need to wrap the synchronous channels in methods so that we can use the bank account from Java code. There is a special utility that creates appropriate methods automatically. We can input

```

void deposit(int amount) {
    this:deposit(amount);
}

```

to describe the action associated with this method. Not all methods will be so simple, e.g., there might be more than one channel invocation.

The translator needs to know other things besides the methods, especially the name of the net, here `account`, and the name of the stub class that should be generated. In this case we use the class `samples.call.Account`. `samples.call` seems to be the right package because the package name should reflect the directory it is in. The full stub definition file can now be presented.

```

package samples.call;
class Account for net account {
    void deposit(int amount) {
        this:deposit(amount);
    }
    void withdraw(int amount) {
        this:withdraw(amount);
    }
}

```



```

    }
    int currentAmount() {
        this:amount(return);
    }
}

```

The declaring package is given in a special statement, which is optional. The keywords **for** **net** separate the class name and the net name.

The body of a class description consists of a sequence of method descriptions and constructor inscriptions. In our example we do not have constructors, such that a default constructor will be automatically inserted. The body of each method consists of a sequence of channel invocations and variable declarations, separated by semicolons.

As in reference nets, variables need not be declared. If variables are declared, they must be declared before they are used. In our example there are no variables except for the input parameters and the special variable **return**, which is used in the last method **currentAmount()**. This variable is automatically declared in each method that has a non-void return type. A non-void method returns the value of **return** at the end of its body.

The stub description can now be compiled with the command

```
compilestub samples/call/Account.stub
```

from the Unix command prompt, assuming that the stub description is contained in the file **samples/call/Account.stub**. A similar script is also provided under Windows, but for other operating systems we do not currently supply a convenient shell script, but you can achieve the same effect by running

```
java -cp plugins/de.renew.misc-2.0.0.jar de.renew.call.StubCompiler \
    samples/call/Account.stub
```

or similar commands. Now the command (the classpath has to be entered on one line!)

```
javac -classpath de.renew.loader.jar:libs/reload4j-1.2.25.jar:plugins/\
    de.renew.util-2.0.0.jar:plugins/de.renew.misc-2.0.0.jar:plugins/\
    de.renew.formalism-2.2.0.jar:plugins/de.renew.simulator-2.3.0.jar samples/\
    call/Account.java
```

compiles the Java source resulting in the file **Account.class**.

Because this command is rather long, we provide the script **jcompile** for Unix and Windows which includes an appropriate classpath to compile Renew-related classes. Just type the following command to achieve the same result:

```
jcompile samples/call/Account.java
```

We will now use this class inside a reference net, but it could be used in Java code just as well. When used in Java code, the generated class extends the **NetInstanceImpl** Class, so all **NetInstance**-methods can be invoked on this class. When creating and using this class in code, there will be no graphical feedback. This point should be kept in mind. The only limitation is that the net assigned to this class has to be loaded in Renew. At the moment, Renew does not provide an automatic net loading mechanism that would correspond to class loading in Java. In Fig. 3.19 you can see the net **customer** that describes a customer accessing a bank account. A new account is created, money is deposited, and the customer checks the current savings.

If you load the two nets from the directory **samples/call** and start the simulation of net **customer**, you will see that the firings of the transitions are no longer sequential. E.g., we have in the simulation log file:

```

...
(3) ----- Synchronously -----

```



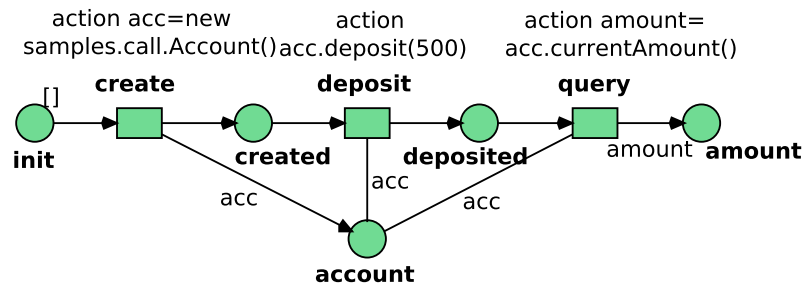


Figure 3.19: The net customer

```

(3) Removing [] in customer[0].created
(3) Testing account[2] in customer[0].account
(3) Firing customer[0].deposit
(4) ----- Synchronously -----
(4) Removing int(0) in account[2].money
(4) Firing account[2].deposit
(4) Putting int(500) into account[2].money
(3) Untesting account[2] in customer[0].account
(3) Putting [] into customer[0].deposited
...

```

The transition `deposit` of `customer` fires at step (3), but at first it can only remove its input token and test the account reference. The output token is not put into the place `deposited` before the action inscription is completed. This requires the invocation of the method `acc.deposit(500)`. Because this method must perform a synchronization, it cannot complete immediately. First, the method requests a synchronization with transition `deposit` of net `account` in step (4). After that step, the method returns, the action is completed and a token appears in place `deposited`.

Note how the individual steps are mixed with each other. Here we have true concurrency in the simulation, because the method is invoked in the background in a separate thread and operates independently of further firings. In fact, actions that include a method call are always executed in the background. But often the search for a new binding takes so much time that the background thread finishes long before the next binding is found.

But here we have a method that requires a synchronous communication before it completes. Such methods rely on the simulator thread to find a matching channel and they require more than one step in any case.

The stub compiler recognizes the declaration `class Name for netinstance` (without a specific net name) as an alternative to the declaration given above. Such a stub class owns a one-argument constructor that expects an existing net instance to be wrapped by the stub object. The stub object can be passed around like any plain Java object. It forwards its method calls to synchronous channels of the wrapped net instance as explained above.

It should be noted, however, that there are two objects with individual identities involved (the stub and the net instance). In contrast, the `for net netname` declaration creates one single object only which *is* both the stub and the net instance.

### 3.9.2 Event Listeners

Nets that implement methods might be useful for designing a graphical user interface where the window system sends events that must be processed by a listener. E.g., a button triggers a `java.awt.event.ActionEvent` that is handled by a `java.awt.event.ActionListener`.

```
public interface ActionListener
```



```

    implements java.util.EventListener
{
    void actionPerformed(java.awt.event.ActionEvent);
}

```

Of course, a net could implement the `ActionListener` interface, but there is a catch. The call to an event listener blocks the entire Java windowing thread, such that no events can be processed before the listener completes the method call. Because further user interactions might be needed to trigger the next simulation step, we might run into a deadlock.

To solve this problem, we may denote that a method should return *before* its synchronous channels are invoked. The channel calls are then processed in the background where they do not block other tasks. Of course this is only possible for `void` methods, because other methods must first compute their return value. We will indicate such methods with the keywords `break void`, suggesting that another thread of control breaks off the main thread.

As an example we will create nets that display a window with three buttons that grow, shrink, and close it. (A similar exercise is given in [19].) The interface `ActionListener` is implemented by:

```

package samples.call;
class SizeChanger for net sizechanger
    implements java.awt.event.ActionListener
{
    SizeChanger(java.awt.Frame frame)
    {
        this:setFrame(frame);
    }
    break void actionPerformed(java.awt.event.ActionEvent event)
    {
        this:putEvent(event);
    }
}

```

The constructor takes one argument, namely the frame whose size should be changed. The single method is designated `break void`, so that it can return before any synchronizations are performed. This stub is contained in the file `SizeChanger.stub` that resides in `samples/call` along with the nets from Figs. 3.20 and 3.21.

The net `buttonmaker` is used to construct the frame with its buttons and the `SizeChanger` object. Later on, each mouse click on one of the three buttons results in an event that is propagated to the `sizechanger` net. Every event is equipped with a command string that determines the action to be taken. It is always a good idea to be able to close a window, because otherwise an undue amount of unclosable windows might accumulate on the desktop.

### 3.9.3 Automatic Generation

A single synchronization per method is only appropriate for the most trivial methods, namely those methods that can be completed atomically. Most methods will require at least two synchronizations, one to pass the arguments of the call and one to collect the results. A very simple scheme would require the following two channel invocations.

```

this:method(arg0,arg1,arg2)
this:result(return)

```

When two or more concurrent method calls are allowed, this scheme breaks up. It becomes impossible to match the two synchronizations and a caller might receive a result that was requested by someone else.

Therefore we consider a more elaborate scheme where each method call is identified by some method instance value.



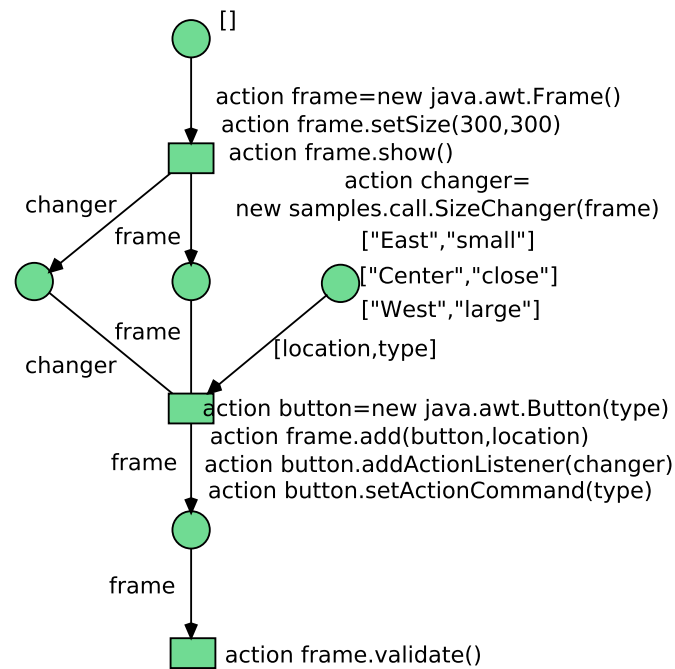


Figure 3.20: The net `buttonmaker`

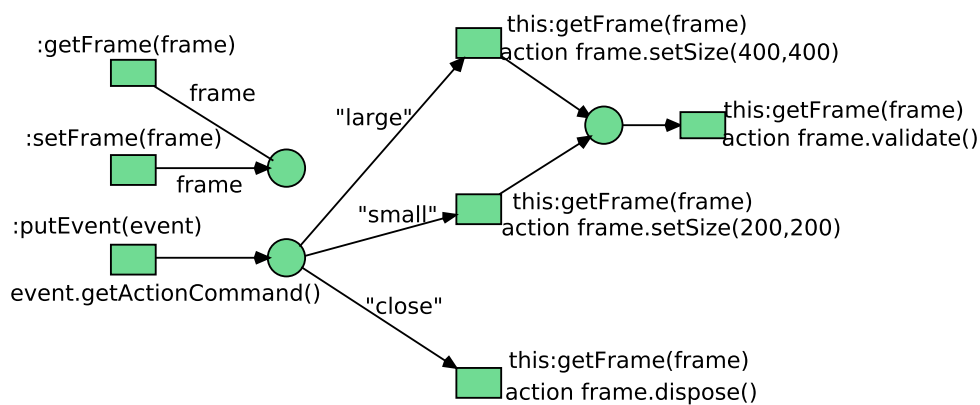


Figure 3.21: The net `sizechanger`



```
this:method(instance,arg0,arg1,arg2)
this:result(instance,return)
```

The first channel provides the arguments to the net and receives a method instance value back. We do not specify how this value is composed, but it must identify the original call uniquely.

In the case of a void method it would not be sensible to compute a **return** value, hence we could leave out the **return** parameter from the second channel invocation. It still makes sense to have the second invocation, though, because we usually want to wait for the completion of the method.

```
this:method(instance,arg0,arg1,arg2)
this:result(instance)
```

There is one problem with that solution, namely that methods should be able to throw exceptions. Because exceptions in Petri nets are not very well understood, we did not implement an exception mechanism right now. It might be added in several ways, none of which looks entirely satisfying.

A regular structure of the synchronization requests suggests that we could generate the stub description files automatically. This is indeed possible using the Unix shell script **makestub** that creates a stub automatically. The script needs the name of the class to be generated, the name of the net that is associated to the class, and a list of interfaces that the class should implement.

For **makestub** the methods that are to be implemented are given only via the list of interfaces. This might seem as a limitation, but quite often appropriate interfaces will be present and in other cases they can be defined easily. And even in ordinary Java it is often helpful to declare all public methods in interfaces.

Assume that Santa's quality assurance department determines that the current version of Santa's bag violates the design rule that bags should implement the `java.util Enumeration` interface. Now a simple command

```
makestub samples.call.EnumBag enumbag java.util Enumeration
```

creates the file `samples/call/EnumBag.stub`. On some non-Unix machine you might have to use the command

```
java -cp plugins/de.renew.misc-2.0.0.jar:plugins/\
      de.renew.simulator-2.3.0.jar:plugins/de.renew.util-2.0.0.jar:. de.renew.\
      call.StubGenerator samples.call.EnumBag enumbag java.util.\
      Enumeration
```

which has the same effect, but is a little longer.

Now the stub file can be compiled as described in Section 3.9.1, i.e., by calling

```
compilestub samples/call/EnumBag.stub
jcompile samples/call/EnumBag.java
```

or equivalent commands.

Fig. 3.22 shows the net associated to the new stub. After `nextElement` is invoked, a new object is created that serves as an identifier for this call. It is also checked that there are still items in the bag before proceeding. An item is taken out of the bag and passed back. The result transition can be shared for both methods, because it simply takes the results from a place and forwards them through the uplink.

Note that the bags are now filled by the manufacturer instead of Santa due to a request of his worker's union. Hence Santa's procedure has to change, too. In Fig. 3.23 you can see Santa distributing the Christmas presents.



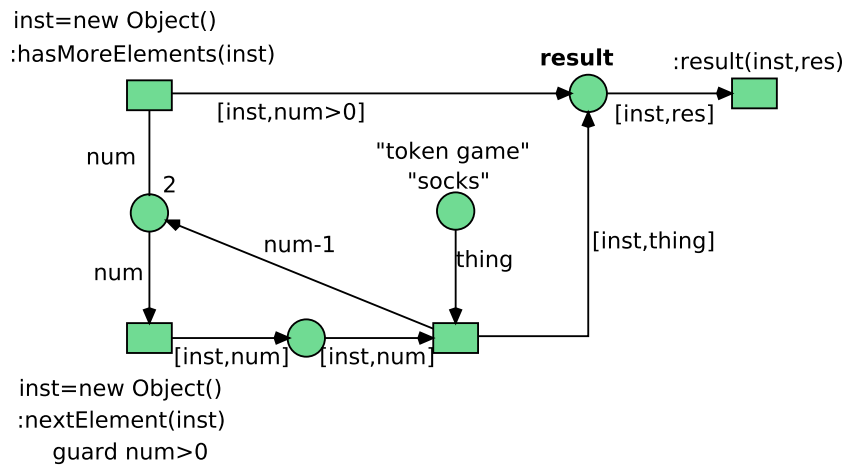


Figure 3.22: The net `enumbag`

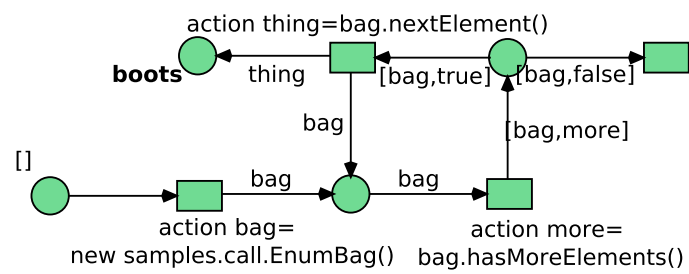


Figure 3.23: The net `enumsanta`



Again, all presents are dropped into the boots over time, but now Santa knows when his bag becomes empty, so that he can fly back and feed his reindeer.

There are actually a few other ways to implement method calls on the level of nets. E.g., one might create a new net instance for each method call and pass the arguments of the call to it. This way the net instance itself could be used to identify the call and the transitions that handle the call could be moved to another net, thereby leading to a much cleaner design. The method net instance could either deposit its result in the common result pool as shown in the previous example, or the result transition could take the result directly from the method net instance by yet another synchronous channel.

## 3.10 Additional Arc Types

Besides those arcs that are commonly found in Petri net simulators, Renew implements a couple of additional arc types that are somewhat rarer, but still quite useful. This is done to achieve the maximum usability for different users by providing adequate modeling tools.

It is sometimes argued that these arc types violate the spirit of true Petri nets. In some sense, they do. Test arcs, too, carry the stigma of bad concurrency semantics. But let's not ban these extensions so rapidly. They have their uses and they can simplify certain models considerably.

If you are unsure about their theoretical foundations, do not use them. No compromises were made in the simulation engine to allow their implementation, so that robustness of Renew is not affected.

### 3.10.1 Flexible Arcs

Flexible input arcs and flexible output arcs were introduced by Reisig in [16]. They allow multiple tokens to be moved by a single arc. Moreover, the token values and even the number of tokens may vary with the binding of the transition's variables.

In Renew, these arcs are indicated by attaching *two* arrowheads instead of one to the end of the arc. In the original article, no such distinction was made, but instead the flexible arcs were indicated by the type of the arc inscription, which is not feasible in the case of reference nets. Besides, flexible arcs do not occur that often and they do deserve some special attention and highlighting.

The inscriptions of flexible arcs must be of an array type. All elements of the array that is calculated from the arc inscription are successively removed from the input place or put into the output place, depending on the direction of the arc. If one Java value occurs multiple time in the array, an equivalent number of corresponding tokens will be removed. The order of the values in the array does not matter.

Arrays are preferred over vectors or other container objects, because they allow the use of primitive values, whereas vectors can only carry references to objects. This can cause some inconveniences to the developer, because occasionally an array is more difficult to generate.

In those cases where it is not feasible to use arrays, Renew support the use of expressions of the types `de.renew.unify.List` (see section 3.5 for details) and `java.util.Collection`. For output arcs, it is allowed to use `java.util.Enumeration` and `java.util.Iterator` objects beside those types listed above. But due to the missing type safety, arbitrary values can be contained in these container objects. Hence, output places for flexible arcs using the container objects must be untyped.

It should be noted that flexible arcs do *not* help the simulator to find information about possible bindings. In some other tools, all possible combinations of tokens are tried for of flexible input arcs, possibly binding variables inscribed to the arc. This was seriously considered, but the performance cost turned out to be prohibitive. At the same time, the need for such an algorithm was not obvious.



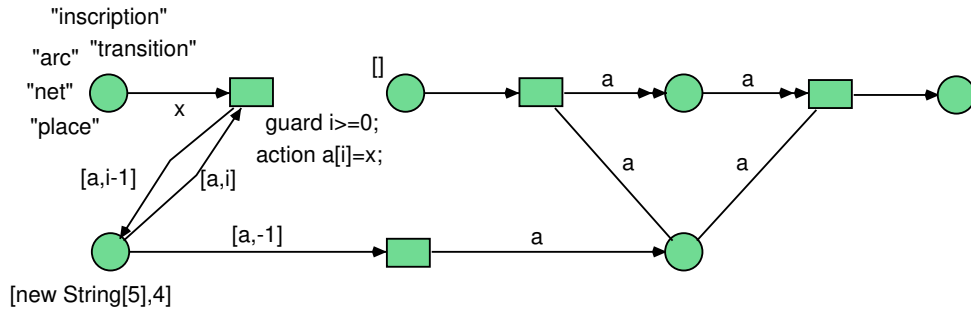


Figure 3.24: The net `flexible`

The net from Fig. 3.24, which can be found in the directory `samples/arcs`, illustrates the use of flexible arcs. On the left hand side you can see a classical way to remove five tokens from a place by looping with an explicit counter. One after another the tokens are collected and assigned to an array, which results in a rather clumsy net structure. Now we can see the two kinds of flexible arcs in action on the right hand side. One transition puts five tokens onto a place and another transition removes all five tokens atomically. The simplification of the net diagram is quite obvious.

Currently, flexible reserve arcs are not supported. They will be added as soon as somebody points out a useful application for them. Flexible test arcs can be added, too, but even their graphical representation is not obvious right now. In general, we reserve the right to make some modification to the handling of flexible arcs, if some other syntactic or semantic variant proves superior.



The current implementation of flexible arcs shows inconsistent behavior when the arc expression does *not* evaluate to an array or collection object as explained above. In such cases, flexible output arcs fall back to shove the unmodified token into the output place, while flexible input arcs do not activate the transition. This is obviously not a symmetric behavior.

In the book [17] Reisig applies flexible arcs to model distributed algorithms. He uses algebraic Petri nets, which are in general not accessible for `Renew`. However, in the given context only very specific algebras are used, namely those that represent communication topologies in a distributed system.

Therefore it is possible to implement an interface `Topology` that captures the signature of the most common algebraic operations and specialized classes that implement some useful topologies, like `RingTopology`, `StarTopology` or `LineTopology`. If further topologies are needed, new classes can easily be added, say for hypercubes or meshes.

The net from Fig. 3.25 shows a simple algorithm that determines a global leader within a network of processors that can only communicate locally. The green net elements constitute the core algorithm as presented in [17]. Additional yellow net elements care for the initialization process. You can see the colors when you load the net from its directory `samples/reisig`. In your nets, too, the use of color might improve the nets' presentation.

In this example, a star topology is used. By default, a star topology is directed, but in this case we add the inverse to the original topology, so that a symmetric topology arises. Flexible arcs are used to send messages to all neighboring arcs whenever a possible new leader has to be announced.

Fig. 3.26 shows the same net, but augmented by a visualization component. For every node of the communication network there is a place that is marked with the node of best currently known priority. Transitions with synchronous channels, which are located in the lower part of the drawing, are used to sort the information into the various places. Virtual



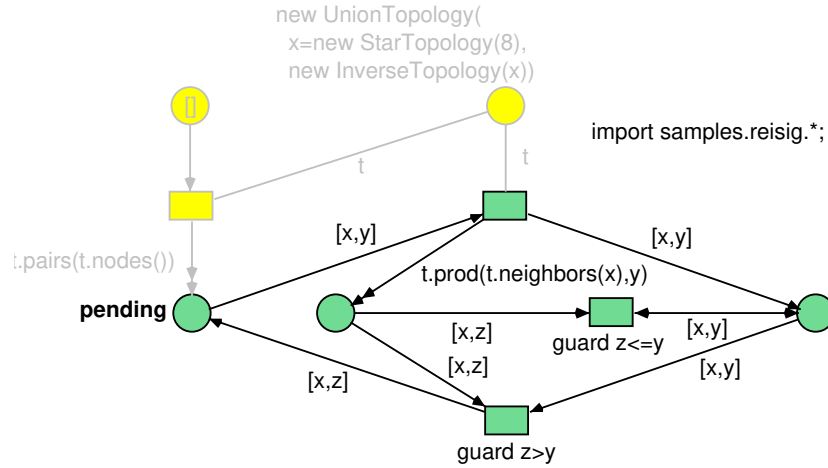


Figure 3.25: The net `election`

place copies, depicted by doubly lined circles on the right hand side, are arranged according to the used topology, so that the topology becomes immediately obvious. Of course, it is now harder to try different topologies, but this kind of visualization is quite effective in classroom demonstrations.

### 3.10.2 Clear Arcs

Clear arcs are used to remove all tokens from a place. They are typically applied to reset the state of the net to a well-defined marking. [12] gives some thoughts on clear arcs and on many other arc types.

In order to use this arc type, you need to select the extended sequential mode as described in Subsection 2.6.4.

A clear arc is indicated by a double arrow tip attached to the transition's end of the arc, where the arrow tip is hollow, i.e., filled with the background color. It need not be inscribed. If it is inscribed, it must be inscribed with a single variable that is either untyped or that has an array type. In the latter case, all the tokens in the place are put into an array and the variable is bound to the array.

Note that the binding occurs only during the firing, i.e., as though the variable was assigned in an action statement. Hence you cannot use the variable of a clear arc for other input arc inscriptions or for guards. However, you can use the variable for output arcs. This is very useful in conjunction with a flexible output arc, when you want to move all tokens from one place to another.

The net from Fig. 3.27 shows the basic algorithm of a juggler who wants to earn a few coins in a crowded mall. He waits for spectators to come along and does a few tricks now and then. This makes all spectators who were previously waiting happy and everybody contributes a fiver. Try to load the net from the directory `samples/arcs` and play around with it.

In the net implementation, note that the variable `x` is bound to an array that holds all tokens of the place `waiting` during the firing of the transition. We use the array to move all the tokens to another place. The array can also be used in ordinary Java computations, if desired. But because the set of tokens is determined only during the transitions firing, a variable attached to a clear can only be used in an `action` inscription, but not in guards, in other transition inscriptions, or in arc inscriptions.

A clear arc takes effect after all other input arcs have been evaluated. This means that you can remove some tokens from a place using ordinary arcs and then remove the remaining tokens with a clear arc. In the given net, you may use this feature by adding an ordinary



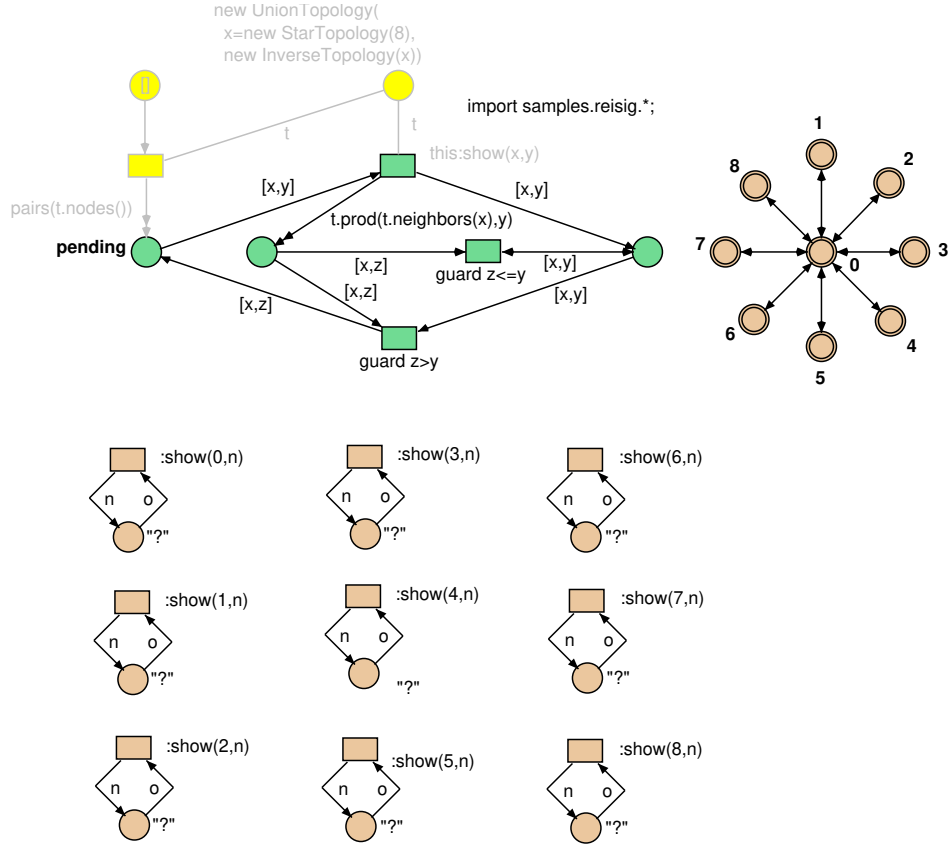


Figure 3.26: The net visuellelection

arc from the place **waiting** to the transition **perform**. This way, the juggler only starts his business when there is at least one spectator.

### 3.10.3 Inhibitor Arcs

Inhibitor arcs make sure that a token of a certain kind is *not* in a place. They are used to represent boolean conditions with simple black tokens when it is required to check for the inverse condition, too. They are also used to delay certain actions until a system is idle and to wait until the end of a loop.

In order to use this arc type, you need to select the extended sequential mode as described in Subsection 2.6.4.

Some varieties of inhibitor arcs were suggested in the literature, see [6] and [12] for a recent approach and [4] for a consideration of concurrency issues. The papers also give further references. The formalism presented here is less general than that presented in [6], but on the other hand we do not require that the place associated to the inhibitor arc is bounded by a capacity.

An inhibitor arc is represented by a connection with a filled circle on each end in Renew. Some other formalisms highlight only one of the line's ends, but we prefer the symmetric appearance because it emphasizes that no changes to the current marking are performed by inhibitor arcs.

An inhibitor arc may be inscribed just like an ordinary input arc. An inhibitor arc, however, does not contribute any information about possible bindings to the simulator, i.e., all variables used in the inscription must be determined by other inscriptions.

It is not possible to inhibit whole groups of token values, e.g., to make sure that no tuples



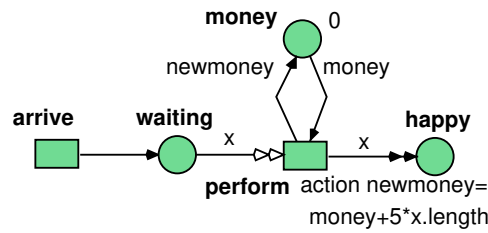


Figure 3.27: The net juggler

whose first component is a given value are contained in a place. But this can usually be simulated by providing a second place that stores only the first components of the tokens and that is updated consistently with the original place.

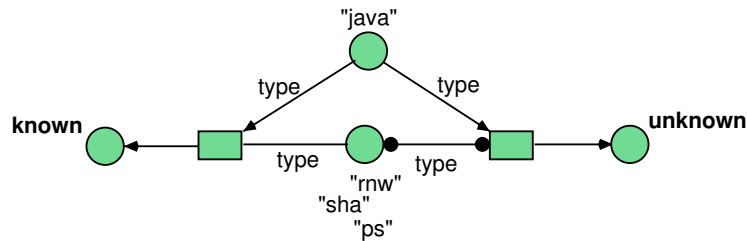


Figure 3.28: The net filetype

In Fig. 3.28 a typical use of inhibitor arcs is shown. In the place at the top, a filetype is given and the system has to decide whether this is a known file type. All known file types are supposed to be registered in the place at the lower center. Now the left-hand transition selects the known file types, whereas the right-hand transition can only fire if the filetype is *not* contained in the center place. Using ordinary Petri nets it is rather difficult to express such a constraint.

Note, however, that inhibitor arcs lack a robust concurrency semantics, so that they have to be used with the extended sequential mode as described in Subsection 2.6.4.

### 3.11 Timed Nets

While pure Petri nets capture the causality and conflict situations of a system nicely, there are reasons to add a notion of time to the formalism in order to model additional dependencies. This is especially true in the case of simulations of physical systems.

To enable timed simulations, two options need to be configured. The usual Java compiler complains about time inscriptions, you have to choose the **Timed Java Compiler** in the **Formalisms** menu (see section 4.3.6) or with the `setFormalism` command (see section 2.7.4). Timed nets can *only* be simulated in the sequential simulation mode as described in subsection 2.6.4. For your convenience, the sequential simulation mode is enabled automatically when you choose the **Timed Java Compiler** in the editor.

In timed mode, a time stamp is attached to each token. It denotes the time when the token becomes available. Delays may be used with arcs in order to control the time stamps of token and the firing times of transitions.

A delay is added to an arc by adding to the arc inscription the symbol @ and an expression that evaluates to the number of time units. E.g., `x+1@t` indicates that the token value `x+1`



has to be moved after  $t$  time units. Input arcs can require that a token remains available for a given time before enabling the transition. For input arcs, the delay must not be created by a random number generator or depend on the result of an action inscription. However, an input arc delay may depend on token values and indeed on the value of the delayed input token itself, which means that  $[x, t]@t$  would be a legal, although somewhat peculiar arc inscription.

Output arcs can specify that a token is only available after some time. The output arc delay may be calculated in an action and it may be a random number. The output arc delay cannot influence the enabling of a transition, but only the timestamps of the generated tokens. Reserve arcs can specify an output arc delay. The input arc delay is always zero, the token is consumed at the current time.

Test arcs cannot specify a time. They can only access currently available tokens. They put the token back with the original time stamp. Clear arcs cannot specify a time. They remove all tokens, regardless of the time stamp. Inhibitor arcs cannot specify a time. They block on all tokens, regardless of the time stamp. Flexible arcs can remove only currently available tokens.

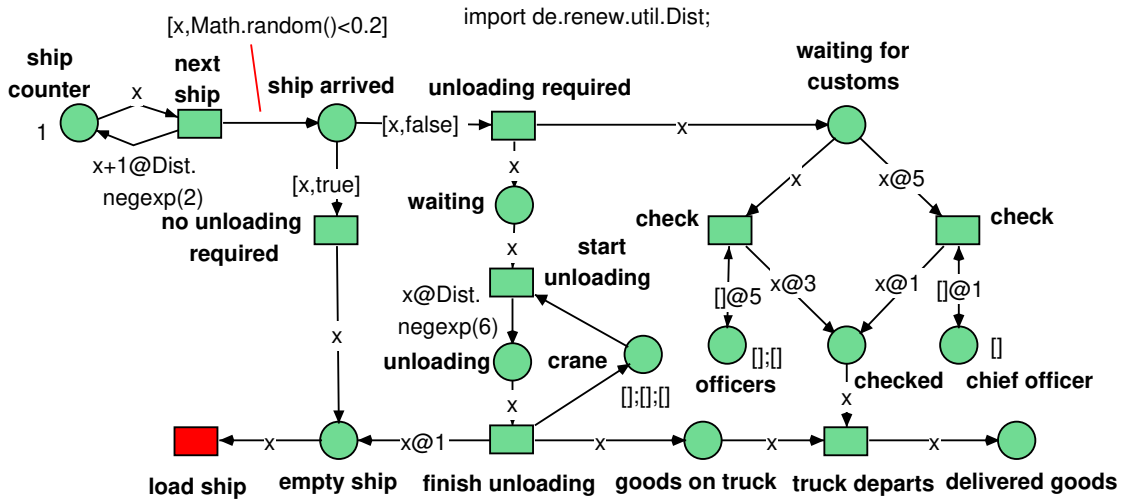


Figure 3.29: The net port

In Fig. 3.29 you can see an example model of a sea port where ships arrive and are unloaded. The loading of the ships is not displayed. At the upper left hand corner, you can see that new ships are created, each ship being numbered accordingly. Probabilistically it is determined whether the ship needs unloading. If yes, the ship is unloaded and the customs declarations are checked concurrently. The ship is unloaded by one of three cranes, where the unloading takes six hours on average with a negative-exponential distribution. Two customs officers handle declarations in three hours, but they need another two hours for filing afterward. If a declaration is not handled for at least five hours, the chief customs officer helps processing the forms. He does not look very carefully and needs only one hour in total. With the good and the signed customs forms, a truck may leave the port. After another hour of cleaning up, the ship may be reloaded.

Note how we use a reserve arc to model a resource that is unavailable during some time. An input arc delay was used to prioritize the ordinary customs officers over the chief officer and to force the chief officer into action at the right time.

You might want to try to add different unloading times to the three cranes. Try to model the loading of the ship. Consider partial unloading of ships. Try to convert the net to an object-oriented design, where the port, the cranes, the ships, and the officers are all individual



nets that communicate via synchronous channels.

## 3.12 Pitfalls

A few common and especially dangerous pitfalls will be discussed in this section.

### 3.12.1 Reserve Arcs and Test Arcs

Reserve arcs and test arcs look alike, because they do not change the marking of the associated place. This can lead to subtle modeling errors.

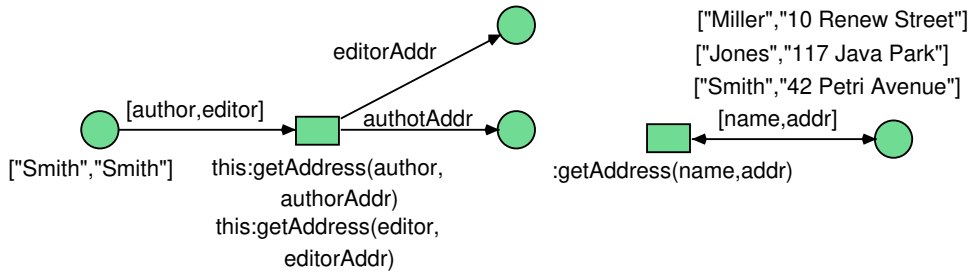


Figure 3.30: The net **reserve**

The net from Fig. 3.30, which is filed in the directory `samples/pitfalls`, shows a small excerpt of a workflow when a printing error is found in an article in a book. The desired effect is to lookup the address of both author and editor, so that they can be sent a notification. The modeler wanted less net elements, therefore both lookups were done by a single transition.

However, the database access is done with a reserve arc, so that this procedure fails when author and editor coincide. In this case, it would have been better to use a test arc, because there is no need to reserve the information in the database. This error, which is not the only one in the net, is especially difficult to detect because it is hidden behind a synchronous channel invocation.

Use test arcs to access information, use reserve arcs to access physical items or logical resources.

### 3.12.2 Unbound Variables

The main task of the simulation engine is to find bindings of the variables under which a transition becomes activated. However, the simulation engine never tries to bind variables blindly to all possible values, e.g., trying -2147483647, then -2147483646, then -2147483645, until a binding of an integer variable is found. Instead, variables that occur as inscriptions to input arcs are bound to the values that occur as tokens in the corresponding places.

This leads to an important design rule: Always make sure that the bindings for all variables can be determined by binding input arc variables to tokens. Remember that the simulator does not evaluate expressions backward. Remember that flexible arcs do not contribute to variable bindings.

If the simulation engine does not manage to bind a variable in this way, it simply gives up and declares the transition disabled. In some other Petri net formalisms, unbound variables are used as a sort of random generator. While this may be a good idea sometimes, it is not difficult to simulate this behavior by a direct call to a random number generator.

Closely related are spelling mistakes for variable names. In the untyped formalism every identifier is a legal variable name, therefore many spelling mistakes cannot be detected. Often this leads to unbound variables and completely disabled transitions, although all tokens seem



to be in place. In fact, Fig. 3.30 contains such an error, because in one place `authorAddr` is misspelled as `authotAddr`.

### 3.12.3 Side Effects

It has already been noted that side effects must only occur in action inscriptions. However, there is another tricky point: The enabledness of a transition must not depend on a mutable property of a Java object.

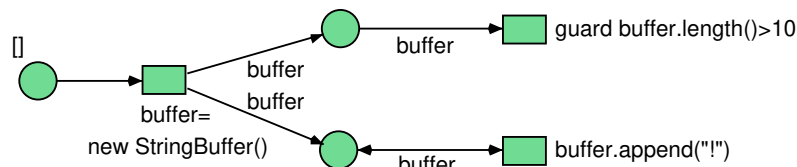


Figure 3.31: The net `buffer`

In Fig. 3.31 we have a net whose modeler is guilty on both accounts. It was intended to have a string that grows and grows and that some other transition should fire as soon as the length of the string exceeds 10. But if you run the net, you will find out that the upper right transition never fires. Or at least it is very improbable that it fires. This is because it is checked for enabledness early on, while the string length is still 1 or 2. Afterward it is not rechecked, because its input place did not change its marking.

It is also possible that the length of the string increases by 2 during some cycles. How can that be? The call to `append` is not contained in an action, so that could be evaluated once during the search for a binding and once during the actual firing. Note that this is a relatively harmless scenario.

### 3.12.4 Boolean Conditions

The standard way of ensuring the equality of two expressions is the transition inscription `expr1==expr2`. But note that the inscription `expr1!=expr2` does *not* lead to inequality. It is merely a boolean expression that is evaluated and whose result is discarded. If you want to specify inequality, you have to use a guard like `guard expr1!=expr2`.

### 3.12.5 Custom Classes

It is often sensible to encapsulate complex operations in helper classes that are associated with a net. In this way, it is possible to keep the nets free of unneeded detail. Of course, the helper classes need some changes occasionally and have to be recompiled.

If the helper class was already used in the simulator at the time of recompilation, e.g. in a previous simulation run, then the Java virtual machine will *not* load it again. Instead it will continue to use the old version of the class. To reload new classes, you either have to close and restart Renew entirely or use the class reinit mode as described in subsection 2.6.5.

### 3.12.6 Net Stubs

A net stub (see 3.9) is created with the name of its associated net. At runtime the net stub tries to find a net with this name, but it will only succeed if the net is already compiled or if the net is accessible through the net loader (see section 2.6.6). If not, the constructor will throw an unqualified `RuntimeException`.

When using a stub declared with `for net netname`, it is important that the stub object is created instead of the net instance itself. This means that the net should not be instantiated



by the usual `:new` inscription. The constructor of the stub class must be used because such a stub is not able to wrap an existing net instance. The alternative declaration `for netinstance` allows wrapping of net instances, but creates a second object.

Calls to stub methods or the stub constructor should be encapsulated in **action** inscriptions because these methods cannot terminate until the simulation step needed for the involved channel synchronization is completed. The simulator may hang if stub methods are executed outside of actions. (This note applies especially to those cases where a net calls a Java method which in turn calls a net stub.)

### 3.12.7 Execution of synchronized Java Code

This comes close to the execution of methods with side effects. The side effect of a synchronized method or code block is that it delays the execution of other threads. This can lead to a deadlock when the thread of the simulation engine has to wait in a monitor for another thread which cannot finish without the simulation thread (if you are asking how to produce such a constellation: Call a net stub from synchronized code, for example).

There are two ways how to avoid the problem:

- Do not use synchronized Java code. Every time you would need synchronization, model it explicitly with nets. This way you are able to *see* the problem, if one occurs.
- When calling a synchronized method, make the calling inscription an **action** inscription. So the simulation engine is out of danger of waiting in a Java monitor during the search for bindings. If you are using the default concurrent simulator, it stays alive and is able to continue its work. It still can happen that the execution of the action inscription hangs, but then the responsible transition is highlighted and can be identified.

### 3.12.8 Case of Class and Variable Names in Untyped Nets

The inscription language of Java reference nets is case sensitive. However, if you create untyped nets (without any variable declaration in the declaration node), there might be an exception – depending on the operating system you are using.

In untyped nets, Renew has to guess whether a particular identifier denotes a class or a variable. It does so by checking for classes with a matching name in the configured class loader, which in turn looks up all file locations in the class path. On operating systems with a case-insensitive file system (e.g. Windows), the class loader might find a class that matches the queried name except for the case. Renew then reports a class with a linkage problem referring to a `NoClassDefFoundError` (or similar).

The problem occurs especially when you use class and variable names that differ only in case. Although this is acceptable in nets with explicit variable declarations, you should avoid this technique in untyped nets.



## Chapter 4

# Using Renew

Renew offers a graphical, user-friendly interface for drawing reference nets and auxiliary graphical elements. The net editor contained within Renew is based upon a Java library called JHotDraw [8]. The basic drawing capabilities are mainly taken over from JHotDraw, while the multi-windowing GUI, the net editor figures and tools, and the image figure tool have been implemented by the Renew team. Still, this manual covers a complete description of all drawing and editing capabilities Renew offers.

### 4.1 Basic Concepts

When working with Renew, you edit so-called drawings. A drawing consists of many drawing elements, called figures. All commands have been grouped into one central window, the Renew window, which contains a menubar, a toolbar, a frame to display drawings, a tab bar to switch between open drawings and a status line (see figure 4.1).

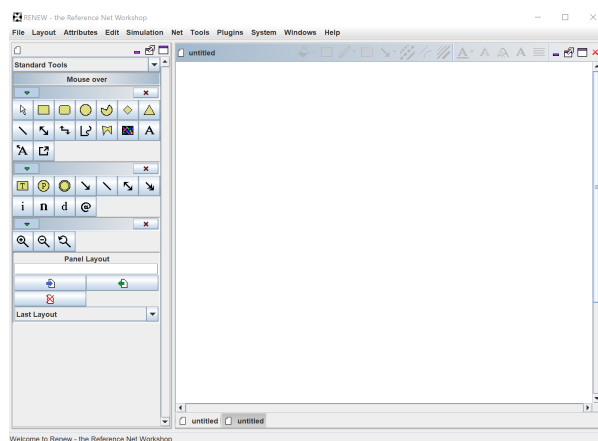


Figure 4.1: The Renew Window

There is always one active drawing window. Selecting a pull-down menu invokes a command which affects the active window, its drawing, or a selection of figures of that drawing, unless it has a global effect only. Examples of menu commands are saving or loading a document or changing attributes of figures. The menu commands are explained in Section 4.3. On the other hand, the toolbar is used for selecting a current tool. With a tool you can create or edit certain kinds of figures in a drawing. All tools available in the toolbar are discussed in Section 4.2. Since each tool (but the selection tool) is related to a certain type of figures, the corresponding figure type is also explained in that section. To manipulate figures, handles



are used. Handles are small squares or circles that appear at special points of a figure when the figure is selected. Dragging and (double-)clicking handles has varying effects, depending on the kind of figure and handle. Handles are also explained in the corresponding figure's section.

To find out how to install Renew, refer to Section 2.4. You should then be able to start Renew from the command line, just typing **renew** (or **loadrenew**), or using a program icon you created, depending on your operation system.

You can also provide some drawings' file names as command line parameters. After typing **renew**, just provide the (path and) name of one or more files, including the extension **.rnw**, e.g.

```
renew MyNet.rnw some/where/OtherNet.rnw
```

On start-up, Renew tries to load drawings from all specified files. On Unix systems, you can even use

```
renew some/where/*.rnw
```

to load all drawings in a directory.

If you have a program icon that is associated correctly, your OS usually also supports double-clicking some **.rnw** file or using drag & drop.

In the rare case that Renew terminates abnormally, it should leave an autosave file for each modified net drawing. Autosave files are typically updates every two minutes. You can detect an autosave file by its file extension **.aut**. Whenever possible, the filename is derived from the main drawing's file name by removing the old name extension **.rnw** and adding **.aut**. If such a file exists already, a random file name of the form **rnw99999.aut** with an arbitrary number is chosen. In order to recover an autosave file, simply rename it, so that it receives the **.rnw** extension.

Renew also leaves **.bak** files that constitute the last version of the file that Renew loaded. Unlike autosave files, these files are overwritten during subsequent runs of Renew.

### 4.1.1 Window Management

With the release of 4.0 a new window management concept was introduced based upon the Docking Frames framework. In older renew versions all windows were completely separate. Now it is possible to arrange windows into tab groups, align them next to each other and minimize them.

The user interface is composed of one main window and optional additional windows each containing one or more panels.

The main window consists of the menu bar and a content area. The content area is composed of one main area and 4 surrounding edge areas. Panels can be placed into the main area or minimized into one of the 4 edge areas.

Each drawing is a panel. There are three possible ways these panels can be arranged. A panel can be in its own window, multiple panels can be grouped together in one window or a panel can be placed into the main window.

In the top right corner of each panel there are four icon buttons to change the panels arrangement:

**Minimize** By clicking on this button the window is minimized. The reference is placed into one of the four edge sections of the main window from which it can be accessed again.

**Dock/Undock** If a panel is attached to the main window clicking this button detaches it and moves it into its own separate window. If the panel is already in its own window clicking this button reattaches it into the main window.

**Maximize/Initial size** Clicking this button will toggle the panel between the maximal possible size and its original size.



**Close** Clicking this button will close the panel. Some panels such as the tool buttons can not be closed.

## 4.2 Tools

In the toolbar, several tool buttons are displayed, which can be selected by clicking on them.

The tool buttons are grouped in two or more toolbars (depending on the mode of Renew). When resizing the Renew window, toolbars are wrapped according to the size of the window. The standard toolbars are the drawing toolbar and the Petri net toolbar. The different bars are shown at the left side of the Renew window. More toolbars can show up based on the chosen formalism.

Figure 4.2 shows the Petri net toolbars at the left part of the Renew window.

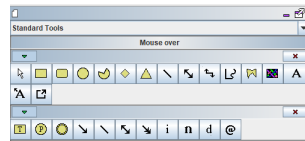


Figure 4.2: The Petri Net Toolbars in the Renew main Window

At any point in time, exactly one tool of all toolbars is selected, which appears pushed down. By default, a special tool, the selection tool, is selected, whenever the work with the current tool is finished.

If you double-click a tool button, the tool will remain active until you explicitly select another tool or right-click on an empty spot in the drawing. This replaces the menu **Toggle Sticky Tools** from the **Edit** menu. In general, double-clicking tools is most useful during the initial creation of nets (but there are also other, probably more elegant ways) and the normal selection is more apt to later modification stages. But of course, which way to use tools also depends on your personal preferences.

In the status line in the Renew window, a short description of the tool is displayed if you move the mouse pointer over a tool button. All other tools but the selection tool are used to create a certain type of figures. Some of the tools can also be used to manipulate already existing figures of the corresponding type.

### 4.2.1 The Selection Tool



The selection tool is the most basic tool and is not related to any special figure type. Instead, any figure or group of figures can be selected and moved using this tool. If not otherwise noted, when talking about pressing a mouse button, the primary mouse button is meant.

If the selection tool is the current tool, the following user interactions are possible:

**Select** By clicking on a figure, it becomes selected. A selected figure can be recognized by its visible handles. Depending on the type of figure, different handles appear, but in all cases, some handles will appear. There are even non-functional handles, which are just there to show that a figure is selected and do not have any additional (manipulation) functionality. If another figure is selected, the current figure becomes deselected. To clear the current selection, click inside the drawing, but not on any figure.

**Add to Selection** If the shift key is pressed while clicking on a figure, the figure is added to or removed from the current selection, depending on its selection status. This way, a group of objects can be selected, which is convenient or even required for some commands.



**Area Selection** If the mouse button is pressed inside a drawing, but not inside any figure, the area selection mode is activated after a short delay. The starting point marks one corner of a “rubber band” rectangle. While the mouse button is held down, the other corner of that rectangle can be dragged by moving the mouse. When the button is released, all figures that are completely inside the rectangle area are selected. Combining this with the “Add to Selection” function is possible.

**Inspection.** Some figures have an additional inspect function that is invoked by double-clicking them, which displays some additional information of the figure without modifying it. E.g., all connected text figures (see Section 4.2.2: The Connected Text Tool) select their parent during inspection.

**Direct Modification** Some figures have an additional direct manipulation function that is invoked by clicking on them with the right mouse button. E.g., all text figures switch into edit mode.

**Dragging** If the mouse button is pressed inside a figure and held down, the drag mode is activated. All figures that are currently selected are moved until the mouse button is released. An advanced feature of dragging is that it is possible to change a figure’s parent. For more information on this function, see Section 4.2.2: The Connected Text Tool.

**Stepwise Movement** You can use the cursor movement keys on the keyboard to move selected figures upward, downward, leftward or rightward in steps of one pixel. If no figure is selected, the cursor keys scroll the viewable area of the drawing in its window. By holding the shift key during cursor movement, selected figures are moved in steps of 10 pixels.

**Manipulating** Depending on the kind of selected figure, handles are displayed at special points within the figure. Using these handles, a figure can be manipulated. The different types of handles are discussed in Section 4.2.2 in the subsection of the corresponding figure’s tool.

**Open Target Location** If the control key is pressed while clicking on a figure, the target location of the figure (if set) is opened (see the subsection of the target tool in Section 4.2.2 for details).

To move a single figure, it is crucial not to hit a figure’s handle, otherwise the handle’s function is invoked instead of moving the figure(s). When more than one figure is selected, the handles of all selected figures are shown but have no effect. To actually use the handles, you have to select exactly one figure. The easiest way to do so is to click on an empty spot in the drawing and then select the figure you want to manipulate.

If you move outside a drawing window while operating with the mouse (i.e. while the mouse button is held down), the viewable area of the drawing is scrolled until the drawing bounds are reached. If you are dragging a figure or handle downward or rightward beyond the current drawing bounds, the bounds are pushed forward until you either release the button or move the mouse back into the window.

In Table 4.1 we summarize the actions of the inspection and direct manipulation functions for all figures. The actions associated to the different figures are explained in more detail in the section that documents the corresponding tool. Some of the entries in the table refer to the simulation mode, which will be explained in more detail in Section 4.3.6.

## 4.2.2 Drawing Tools

Renew provides several drawing tools which create and manipulate drawing figures. These drawing figures do not have any semantic meaning to the net simulator, but may be used



type of figure	double click	right click
rectangle, ellipse, ...	select/select children	select/drag
text	select	text edit
connected text	select parent	text edit
transition	select/select children	attach inscription :s()
virtual transition	select associated transition	attach inscription :s()
place	select/select children	attach inscription []
virtual place	select associated place	attach inscription []
arc	select/select children	attach/edit inscription
declaration	select	text edit
inscription, name, label	select parent	text edit
transition instance	open binding window	fire arbitrary binding
place instance	select marking	open current marking window
cardinality marking	select place instance	show token marking
token marking	select place instance	show cardinality marking

Table 4.1: Summary of selection tool operations

for documentation or illustration purposes. You may lack some functions that you are used to from your favorite drawing tool (like adjusting line width and such), but remember that Renew is a Petri net tool, not a drawing tool in the first place.

### The Rectangle Tool



The rectangle tool is used for creating new rectangle figures. Press the mouse button at the point where the first corner is supposed to be and drag the mouse to specify the opposite corner while holding down the mouse button. While dragging, you can already see the rectangle's dimension and location which is confirmed as soon as you release the mouse button.

After a new figure has been created, the new figure is not automatically selected. To do so, just click on the figure with the selection tool (see Section 4.2.1). Now, the figure's handles appear. In the case of a rectangle or ellipse figure, these are sizing handles which are displayed as small white boxes at the corners of the figure. These handles let you change the dimension (and location) of a figure after you created it. Depending on the position of the handle, only certain changes are allowed. For example, the "east" sizing handle only allows to change the width of a figure, while maintaining the location of the left side, and the "south-west" sizing handle only lets you relocate the lower left corner of a figure, while maintaining the location of the upper and right side. The "south-east" handle restricts itself to sizes of equal height and width (squares) as long as the control key is pressed. With the shift key pressed, the "south-east" handle restricts itself to constrain the proportions. The control key can also be used as a modifier while you are working with the rectangle creation tool (and many other figure creation tools).

All newly created figures have a black outline and aquamarine as the fill color (if there is any). To change these attributes, use the **Attributes** menu (see Section 4.3.4).



To create figures with the same attributes as an existing figure, use copy & paste (see Section 4.3.2).

### The Round Rectangle Tool



The round rectangle tool works the same way as the rectangle tool (see above), only that the created figure is a box with rounded corners. A round rectangle figure has the same handles as a rectangle figure plus an additional single round yellow handle to change the size



of the curvature. Drag this handle and change your round rectangle to anything between a rectangle and an ellipse.

### The Ellipse Tool



The ellipse tool works the same way as the rectangle tool (see above), only that ellipses are created within the given rectangle area. An ellipse figure has the same handles as a rectangle figure.

### The Pie Tool



The pie tool works the same way as the rectangle tool (see above), only that segments of ellipses are created within the given rectangle area. A pie figure has the same handles as a rectangle figure, with two additional “angle” handles that are small yellow circles. The “angle” handles control start and end of the arc segment that frames the pie. By pressing the control key while dragging these handles around, their movement can be restricted to steps of 15 degrees. If a pie’s fill color is set to “none”, it displays as an open arc segment instead of a closed pie.

### The Diamond Tool



The diamond tool works the same way as the rectangle tool (see above), only that diamonds are created within the given rectangle area. A diamond figure has the same handles as a rectangle figure.

### The Triangle Tool



The triangle tool works the same way as the rectangle tool (see above), only that triangles are created within the given rectangle area. A triangle figure has the same handles as a rectangle figure, with an additional “turn” handle that is a small yellow circle. This handle lets you choose the direction the triangle points to, which is restricted to one of the centers of the four sides or one of the four corners.

### The Line Tool



The line tool produces simple lines that are not connected (see also the next section: The Connection Tool). Creating a line is similar to creating a rectangle: Press the primary mouse button where the starting point is supposed to be and drag the mouse to specify the end point while holding down the mouse button.

The line figure has two sizing handles (small white boxes) in order to let you change the starting and end point afterward. It also has an *intermediate point* as described in Section 4.2.2: The Connection Tool.

A line figure has no fill color, but it respects the pen color (see Section 4.3.4).

### The Connection Tool



This tool lets you create connections (arcs) between other figures. A connection is like a line, except that it connects two existing figures and is automatically adapted every time one of the connected figures changes.

Consequently, the location of pressing down the mouse button does not specify a starting point, but a starting figure. Again, the mouse button has to be held down while dragging the end point of the connection. If an appropriate figure is found under the mouse button, the end point “snaps” into this figure’s center. This figure is confirmed as the end point figure as soon as you release the mouse button. The connecting line always is “cut off” at the outline of the start and end figure, so that it just touches their borders.



A connection can be re-connected using its green square connection handles. Just drag one of these handles to the new start or end figure. If you release the mouse button while the connection is not “snapped” into a new figure, the connection will jump back into its old position.

An advanced feature is to produce *intermediate points* (or “pin-points”) in a connection. When selected, connection figures show additional *insert point* handles to create new intermediate points in the middle of each line segment. These are depicted as small circles with a cross (plus-sign) inside. When you click on an insert point handle, a new location handle (see below) is created within the given line segment and can immediately be moved. By holding Ctrl while pressing, holding and dragging an intermediate point you can orient the two emerging line segments in a right angle.

A different method to create and delete intermediate points is to use the connection tool. Activate the tool and click on a point on the connecting line. Now, a new location handle (white square) is created, which you can see the next time you select the connection figure. This handle can be dragged to an arbitrary position.

When you hold down the control key while moving a location handle, the intermediate point jumps to the closest position so that the adjacent line segments form a right angle.

You can also keep the mouse button pressed down right after clicking on an intermediate point and drag the new handle immediately (without actually having seen the handle itself). If you want to get rid of a pin-point, simply select the connection and double-click the associated handle. Another (more complicated) way to remove intermediate points is to select the connection tool and click on the intermediate point with the left mouse button.



If you move two figures, a straight connection is automatically moved with them. But if the connection has intermediate points, these stay at their old location. Solution: Just select the connection itself additionally, and everything will move together.

## The Elbow Connection Tool



The elbow connection tool establishes a connection between two figures just like the connection tool. The difference is that an elbow connection does not draw a direct line from one figure to the other, but uses straight (horizontal or vertical) lines only. When you select an elbow connection, you see up to three yellow handles which adjust the position of the horizontal and vertical lines.



Changes to these handles are not stored. Also, if the connected figures are close together, the decision whether to go horizontal or vertical first is quite poor. Since no elbow connections are needed to construct reference nets, we do not really care about these bugs.

## The Scribble Tool



With the scribble tool, you can effortlessly add freehand figures to your drawings using the mouse. There are two methods available to create a new figure using the scribble tool. Firstly, you can hold down the left mouse button and drag it across the canvas. As you move the mouse, the scribble tool will trace your movement and add points at the current position, gradually forming a shape. Alternatively, you can single-click on the canvas to add individual points, building a figure with discrete points.

The scribble tool offers two options that can be accessed by right-clicking the scribble tool button. Upon right-clicking, a window will appear, allowing you to select between the available options. The first option is "Raw," which adds points based on the mouse movement. These points are then connected by lines, accurately reproducing the shape as you moved the mouse. The second option is "Smooth," which filters the points first and then connects them with curves, resulting in a smoother shape. You can adjust the smoothness level, with higher



values producing smoother shapes with fewer points. However, note that the resulting shape may differ more from the original hand-drawn shape when using higher smoothness settings.

To exit the creation mode of the scribble tool, simply right-click while the scribble tool is selected. Once you have created a figure using the scribble tool, you can still modify individual points by dragging their corresponding white square handles. If you wish to move the entire figure, start by dragging a line segment instead of an individual handle. Alternatively, you can deselect the figure first and then drag it anywhere along a line segment of the figure.

## The Polygon Tool



A polygon is created analogous to a scribble figure (see above). While you create the polygon, you can already see that the area surrounded by the lines is filled with the fill color. In contrast to the scribble figure, the surrounding line is closed automatically. By intersecting the lines, you can create un-filled areas. Like in the scribble figure, there are white, square handles to drag every single point of the polygon figure. A point that is dragged to somewhere on the direct line between its ancestor and predecessor point is removed from the polygon. Also, there is a round, yellow handle that can be used to turn and to scale the entire polygon figure by dragging the handle, which looks really nice (thanks to Doug Lea).

The round, yellow handle is restricted to pure rotation as long as the shift key is pressed and to pure scaling as long as the control key is pressed. The behavior of white, square point handles can be modified with the control key similar that of location handles of connections (see above).

It can be configured how the polygon smoothness lines by removing intermediate points. The property `ch.ifa.draw.polygon.smoothing` can be set to the following values (changes take effect the next time a polygon is manipulated):

**alignment** This is the default behavior. Points are removed if they are located on a straight line between their adjacent points.

**distances** Points are removed only if they are too close to each other (less than 5 pixels distance horizontally *and* vertically).

**off** No smoothing at all (no points are removed).

## The Image Tool



The image tool offers you the possibility to include bitmap graphics into your drawings. When activating this tool, a file dialog box opens that lets you choose a bitmap graphic file from your file system. `gif` files work on all platforms, as well as other formats like `jpg`. Java (and thus Renew) even supports transparent GIF images.



Be aware that the Encapsulated PostScript output does not support transparent GIF images, but some of the other export formats (e.g. PDF and SVG) do.

After you confirmed the file selection, the dialog disappears and leaves you with two options: Either you just click somewhere in your drawing, or you drag open an area, just like when creating a rectangle. If you just click, the image is inserted using its original dimensions (in pixels), otherwise it is scaled to the rectangle area specified by your drag operation.

An image figure has the same handles as a rectangle figure.

Since Renew 2.5 you can use drag and drop to add images. Just drag the image into the drawing editor.

## The Text Tool



The text tool is used to arrange text with your graphical elements. The first mouse click after activating the tool selects the upper left corner of the text area and invokes a text editor.



Now you can type in any text, including numbers, symbols, and so on. You can even use the cursor keys, delete any characters, select some part of the text with the mouse and so on, like in any other Java edit field. Note that you can even type in several lines, as usual by pressing the return or the enter key. This is why pressing return or enter does not end the edit mode.

After you click somewhere outside of the text editing box, the text is entered and all of the text is displayed. If the editing box is empty at that moment (the entered text comprises white spaces and line breaks only), the text figure is automatically removed from the drawing.

The white box handles are just to show that a text figure is selected. The dimension of a text figure can not be changed, as it only depends on its text contents and font selection. The only handle to modify a text figure is a small yellow round font sizing handle. It can be dragged to alter the font size, which can also be done using a menu command (see Section 4.3.4).

If you want to change the text contents of an existing text figure, just make sure the text tool is activated and click on the text figure. The text editor described above will appear. Again, confirm your changes by clicking somewhere outside the editing area.



A fast way to enter text edit mode for any text figure (including connected text, inscription, name, and declaration figures) is to right-click on these figures. The corresponding tool is activated and the figure is put into text edit mode immediately.

### The Connected Text Tool



Connected text works exactly like normal text, except that it is connected to some other figure, which is called its parent.

To create a connected text figure, select the connected text tool and click on the figure that is to become the parent of the new connected text figure. If you select a figure that cannot take a connected text figure or if you select no figure at all, your selection is ignored. If the figure was successfully chosen, continue with editing text like with a normal text figure (see above).

Now, every time you move the parent figure, the connected text figure will move with it. Only when you drag the connected text figure itself, the offset to its parent is changed.

To verify which figure is the parent of some connected text figure, double-click on the connected text figure, and the parent (if there is any) is selected.

A special feature of connected text is dragging a single connected text figure, or any special subclass like inscriptions (see Section 4.2.3: The Inscription Tool), to a new parent. Whenever the “landing point” of a connected text drag operation is another potential parent, it is selected immediately to indicate that instead of changing the offset to the old parent, the targeted figure will become the new parent of the connected text figure as soon as you release the mouse button. If you drag the connected text figure to a location outside this new parent again, its old parent (if there is any) is selected in the same manner, to indicate if you let go the mouse button now, the parent will stay the same.

Note that the offset the connected text figure had to its old parent is re-established for its new parent, so it might jump to another position after reconnection. This is quite convenient if you moved an inscription to a preferred offset to its parent (e.g. to the right-hand side of a transition), and want to keep this offset even after connecting it to a new figure.

### The Target Tool



The target tool can be used to add hyperlinks (target locations) to figures. These target locations can point to other Renew drawings, files in the file system or to a locations written as URI (e.g. a website). The target tool works like the connected text tool with the difference that the target location is only visible when edited. The target location is stored as attribute of the figure (`targetLocation`). The target locations can be opened by using the selection tool and clicking on the figure while pressing the control key (see Section 4.2.1).



### 4.2.3 Net Drawing Tools

Now it is really getting interesting: This group of tools allows you to draw Petri nets that have a semantic meaning to the simulation engine. Renew differentiates between a simple rectangle and a transition, although they may look the same. When you use the net drawing tools, some syntactic constraints are checked immediately (see Section 4.6).



Since all net element figures (transitions, places, and arcs) may have inscriptions, Renew supports automatic inscription generation. Click on a net element figure with the right mouse button, and a new inscription figure is created with a default inscription depending on the type of net element. This is especially convenient for arc inscriptions, since these usually consist of a single variable. Of course, in most cases, you have to change the inscription afterward, but you do not need to use the inscription tool. Instead, you right-click on the net element and then right-click on the newly created inscription.

#### The Transition Tool



This tool functions almost exactly like the rectangle tool. The differences are:

- Only transition figures have a semantic meaning to the simulator. A rectangle figure is ignored by the net execution engine.
- To create a transition with a default size, after selecting the transition tool, just click instead of drag. The position of the click specifies the center of the newly created transition.
- A transition figure offers an additional handle. The arc handle, a small blue circle in the middle of the figure, can be used to create new output arcs (see Section 4.2.3: The Arc Tool).

The new handle has a special behavior when you stop dragging on figure that is not appropriate as a target for the arc. A normal connection is deleted when there is no appropriate end figure. However, for an arc it is quite clear what kind of figure is supposed to be there: a place figure. And this is what happens: Automatically, a place figure is created with its center set to the location where you released the mouse pointer, and the newly created arc connects the transition and the new place.



This feature offers you a very fast way to create reference nets. Just start with a transition and use its blue arc handle to create a new arc and the next place. Since this works for places (see below), too, you can continue to create the next arc and transition using the arc handle of the newly created place. If you want to reuse an existing place or transition, just drag the arc to that figure as usual. Thus, you can create arbitrarily complex nets without selecting any other tool! If you combine this with the automatic inscription generation and editing (see above), even colored nets will only take seconds to create.

#### The Virtual Transition Tool



In order to use virtual transitions, set the property `de.renew.gui.virtualTransition` to `true` (see Section 2.6.1).

You create a virtual copy of a transition by activating the virtual transition tool, then clicking on the transition you want to copy (this can also be a virtual transition!) and keeping the mouse button down while dragging the virtual transition figure to its destination location. The virtual transition can be distinguished from a normal transition by the double border line (see the graphics inside the tool button). To find out which transition a virtual transition belongs to, just double-click the virtual transition. To make this relation visible in printed versions of your nets, you should copy the name of the transition to the virtual transition.



Unfortunately, the tool does not take care of the names of virtual transition automatically. Another solution supported by the tool is to give each group of a transition and all its virtual copies a different fill or pen color. All transition belonging together will change their colors if you change the color for one transition.

During simulation, a transition and all its virtual copies are treated as a single transition. Virtual transitions have the same input set and output set as their semantic transition. This also applies to inscriptions; they are applied to all virtual/semantic versions of the inscribed transition.



Use virtual transitions with care, as you may not be able to see that one transition is connected to another during simulation. You can use synchronous channels to make connections between transitions explicit.

### The Place Tool



The place tool works analogously to the transition tool, only that the arc handle (the small blue circle) creates input arcs (see previous section). If the new arc is not released on top of an existing transition, a new transition is created and used as the target of the arc.

### The Virtual Place Tool



The virtual place tool is used to create virtual copies of a place analogously to the virtual transition tool.

During simulation, every virtual copy of a place contains exactly the same token multiset as its original place. Still, it is possible to determine the marking appearance separately for each virtual place (and the place itself) (see Section 4.3.6).



A nice way to take advantage of this feature is to create virtual copies of places with an important and extensive marking and move these to an area outside the net. This has a similar effect as the current marking window, but you do not get your screen cluttered with so many windows.

### The Arc Tools

The arc tool works quite the same as the connection tool (see description in Section 4.2.2). The differences are, like above, that an arc has a semantic meaning to the simulator. A restriction coming from the Petri net structure is that an arc always has to *connect one transition and one place*, not two figures of the same kind or any other figures. The arc will not snap in on the wrong figures and disappear if you release the mouse button over a wrong figure. This behavior is different from when you create arcs using the arc connection handle in places or transitions (see Section 4.2.3: The Transition Tool).

There are four arc tools for those different arc types that are generally available:

**Arc Tool** – This tool is used for creating **input** and **output arcs**, which only have one arrow tip at their ending. If the start figure of the connection is a place (and thus, the end figure has to be a transition), this one-way-arc is an input arc. If the start figure is a transition, we have an output arc.



**Test Arc Tool** – Here, **test arcs** without any arrow tips are created. A test arc has no direction, as no tokens are actually moved when the transition fires (see Section 3.12.1). This means it does not matter whether you start a test arc at the place or at the transition.



**Reserve Arc Tool** – With this tool, **reserve arcs** with arrow tips at both sides are created. Again, the direction does not matter. For the semantics of reserve arcs, see Section 3.12.1.





**Flexible Arc Tool** – An arc with *two* arrow tips on one side is created. These **flexible arcs** transport a variable number of tokens. For the semantics of flexible arcs, see Section 3.10.1.



There are two additional arc tools that are only displayed on request, as described in Subsection 4.3.6.

**Clear Arc Tool** – This tool is used for creating **clear arcs**, which remove all tokens from a place. You have to select the place as the start figure and the transition as the end figure during the creation of a clear arc. For the semantics of clear arcs, see Section 3.10.2.



**Inhibitor Arc Tool** – This tool is used for creating **inhibitor arcs**, which stop the attached transition from firing as long as certain tokens are contained in a place. This arc features circles at both of its end points. Again, the direction does not matter. For the semantics of inhibitor arcs, see Section 3.10.3.



Using the **Attributes** menu, it is possible to change the direction of an arc after its creation. Simply select the desired value for the attribute **Arrow**. However, you cannot currently change ordinary arcs to flexible arcs, or vice versa. Neither can you access inhibitor or clear arcs this way.

Let us repeat from Section 4.2.2 that you can create intermediate points by selecting an arc tool before clicking on an already existing figure. You can then drag the intermediate point to its destination. To get rid of intermediate point, right-click the associated handles.

## The Inscription Tool



Inscriptions are an important ingredient for most high-level Petri net formalisms. An inscription is a piece of text that is connected to a net element (place, transition, or arc). Refer to Section 3 to find out what kind of inscriptions are valid in our formalism. You can inscribe types and initial markings to places. You can provide inscriptions for arcs, in order to determine the type of tokens moved. Transitions may carry guards, actions, uplinks, downlinks, and expressions. Multiple transition inscriptions may be given in a single figure, but they have to be separated by semicolons.

When editing inscription figures, you have to know that in principle they behave like connected text figures. This means that all functions for connected text figures also work for inscription figures (see Section 4.2.2: The Connected Text Tool). For example, to check that an inscription figure is in fact connected to the net element you want it to be connected to, double-click on the inscription figure. Then, the corresponding net element should be selected. Also, you can drag an inscription to another net element.

Again, in contrast to text figures, inscription figures have a semantic meaning to the simulator. By default, inscriptions are set in plain style, while labels (text without semantic meaning) are italic. The syntax of an inscription is checked directly after you stop editing it (see Section 4.6). Refer to Chapter 3 for a description of the syntax of Renew net inscriptions.

## The Name Tool



The name tool also connects text to net elements, in this case to places and transitions only. By default, a name is set in bold style. The idea of a name for a place or transition is to enhance readability of the net as well as simulation runs. When a transition fires, its name is printed in the simulation trace exactly like you specified it in the name figure. Place names are used in the simulation trace whenever tokens are removed from or put into a place. Also, a place's name is used in the window title of current marking windows and a transition's name is used in the new transition binding window (see Section 4.3.6).

Each place and transition should have at most one name figure connected and each name should be unique within one net (but the editor does not check either of these conditions).



Places and transitions without connected name figures are given a default name like P1, P2, ... and T1, T2, ...

### The Declaration Tool



A declaration figure is only needed if you decide to use types (see Section 3.4.2). Each drawing should have at most one declaration figure. The figure is used like a text figure, only that the text it contains has a semantic meaning to the simulator. The text of the declaration figure is used for import statements as well as variable declarations (see Section 3.4.2).

As in the case of inscriptions (see above), the content of a declaration figure is syntax-checked as soon as you stop editing. For an explanation of syntax errors that may occur, refer to Section 4.6.

### The Comment Tool



The comment tool connects comment texts to net elements. Comment texts have a blue text color as default and no semantic meaning for the simulator.

### The Zoom In Tool



The zoom in tool allows the user to zoom into the picture.

### The Zoom Out Tool



The zoom out tool allows the user to zoom out of the picture.

### The Zoom Reset Tool



The zoom reset tool resets the zoom factor to 100 percent.

### The Zoom Area Tool



## 4.3 Menu commands

This section contains a reference to Renew's menus and the functions invoked by them.

### 4.3.1 File

As usual, the file menu contains every function that is needed to load, save and export drawings. In the following section, all menu items of the file menu are explained.

#### New Net Drawing (\*.rnw)

This menu invokes a function that creates a new drawing and opens it in a drawing window in a default window size. The new drawing is named "untitled" and is added to the list of drawings in memory (see Section 4.3.7).

The keyboard shortcut for this function is **Ctrl+N**.

#### New Drawing...

Renew supports different kinds of drawings (dependent on the installed plug-ins), this menu entry opens a dialog where the type of drawing can be chosen. Select the appropriate drawing type in the dialog and press the **New** button.



## Open Navigator

This command opens the Renew file navigator in a new window. The navigator displays folders and their Renew-related content in a directory tree. The navigator is shown in Figure 4.3.

The keyboard shortcut for this function is **Ctrl+Shift+N**.

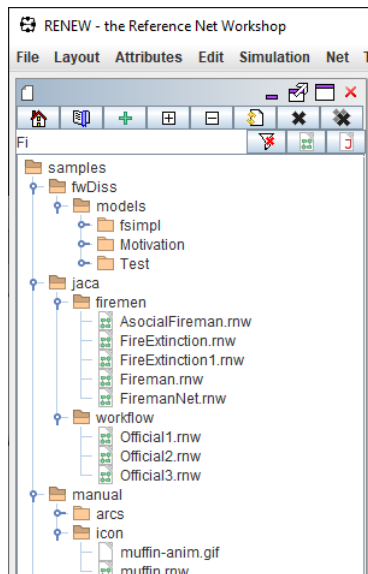


Figure 4.3: The Renew Navigator

**Usage of the Navigator** At the top of the navigator window is an icon bar with eight buttons and an additional filter bar with an input field and three additional filter buttons. We describe these buttons and their function “left to right”.

The **Home** button displays the home directory which defaults to the preconfigured files (see next paragraph) or, if there are none, to the current directory. The **NetPath** button displays all folders which are included in the netpath of Renew. This is usually empty but can be set when starting Renew or in the menu *Simulation* → *Configure Simulation* → *Net Path*. The **Add Files** button opens a file choose dialog and adds the chosen directory or file to the tree. Files and folders can also be added to the Navigator using drag and drop. The **Expand** button expands the complete folder structure. The **Collapse** button collapses all nodes of the tree. The **Refresh** button checks for new and deleted files and updates the display in the tree area. The **Remove** button removes a single node from the tree, while the **Remove all** button removes the whole tree.

The input field can be used to filter the Navigator’s content. The first button next to the input field clears the input field and the last two buttons provide predefined filters for **.rnw** and **.java** files.

The Navigator is persistent and extensible. We optionally provide some convenient extensions, such as the integration of the drawing’s diff feature (ImageNetDiff), which can be triggered directly from the Navigator GUI. The persisted state of the Navigator is saved in the file `navigator.xml` in the `.renew` subdirectory of your home directory.

**Configuring the Navigator** The navigator has two properties that can be configured in the usual configuration files: `de.renew.navigator.workspace` and `de.renew.navigator.filesAtStartup`. The first property defines the base directory for the navigator plugin. It needs to be an absolute path like `/path/to/renew5.0-Pre/` or `c:/path/to/Renew5.0-Pre/`. The second property is a semicolon<sup>1</sup> separated list of paths relative to the base directory. All folders and files defined in this list will be added to the tree area on startup.

Example: `MyNets;Core/samples;../../../../home/renewuser/exampleNets`

## Open URL...

Renew can load drawings from a remote location specified by a URL. This command opens a dialog where you can type the URL and press **OK**. Note that Renew is not able to save drawings to URLs, it still requires a local file name.

<sup>1</sup> A semicolon has to be used even on Unix-based systems, where paths are usually separated with the colon (:).



### Open Drawing...

This function displays a file selector dialog that lets you select a drawing that was saved previously. The file selector dialog looks a little bit different depending on the platform, but always allows you to browse the file system and select an existing file. By pressing the OK button, the selection is confirmed and Renew tries to load this file as a drawing. If that does not succeed, an error message is displayed in the application log and in the status line of the Renew window. Otherwise, the drawing is added to the list of drawings in memory (see Section 4.3.7) and opened in a new drawing window. The keyboard shortcut for this function is **Ctrl+O**.

The open dialog accepts the selection of multiple files. This will result in multiple drawing windows to be opened in the editor simultaneously.

Dependent on the set of installed plug-ins (and on your window manager), one of several available drawing file types can be chosen from a drop down box in the dialog. This will restrict the display of files in the dialog. You may override the file type filter by specifying a wildcard pattern like **\*.\*** as file name and pressing **Enter**.

Alternatively to using the Open Drawing command in the menu, you can use drag and drop to open drawings. Just drag the files into the Renew menu and tools window.

### Insert Drawing...

This function opens a previously saved drawing to be inserted into the currently focused drawing editor (Opening works like in **Open Drawing...**). All figures which are selected before insertion are deselected. In return all the inserted figures are selected now which makes it easy to move them around without jamming the other figures.

### Save Drawing

This function saves the active drawing (see Section 4.1) to a file using a textual format. The drawing is saved to the last file name used, which is the file it was loaded from or the file it was last saved to. If the drawing has not been saved before, this function behaves like **Save Drawing As...** (see below).

If there is an old version of the file, it is overwritten. Depending on your operating system, overwriting a file might need confirmation by the user (you).

The keyboard shortcut for this function is **Ctrl+S**.

### Save Drawing As...

This function is used to determine a (new) file name for a drawing and save it in textual format (see above).

Like in **Open Drawing...**, a file selector dialog is displayed to let you determine the (new) file name and location. After confirming with the OK button, the specified file name is used to store the drawing now and during future invocations of **Save Drawing**. The name of the drawing is set to the file name without path and file extension. If you cancel or do not select an appropriate file name, the drawing will neither be saved nor renamed.

Dependent on the set of installed plug-ins (and on your window manager), one of several available drawing file types can be chosen from a drop down box in the dialog. The effects are similar to the effects in the **Open Drawing** dialog explained above. However, the list of available file types is restricted by the type of the drawing you are going to save.

The keyboard shortcut for this function is **Ctrl+Shift+S**.

### Save All Drawings

This function saves all drawings that are currently in memory (see Section 4.3.7). Before this can be done, all untitled drawings have to be given a (file) name, which is done as in



**Save Drawing As...** (see above). If you cancel any of the save dialogues, no drawing will be saved. If all drawings are given a proper (file) name, they are all saved. You should invoke this function before you exit Renew (see below).

### Close Drawing

Closes the active drawing window and removes the corresponding drawing from the list of drawings in memory (see Section 4.3.7).

Before doing so, Renew checks if the drawing could have been changed (this check is a little bit pessimistic) and if so, asks you whether to save the drawing. You have the options to answer **Save now**, **Close**, or **Cancel**. **Save now** tries to save the drawing. Drawings which already have a name are saved with that name. If the drawing is untitled, the normal save dialog appears (see above). Here, you still have the option to cancel, which also cancels the closing of the drawing. If you select **Close**, the drawing is closed and all changes since the last save are lost (or the whole drawing, if it was still untitled). Last but not least, you have the option to **Cancel** closing the drawing.

The keyboard shortcut for this function is **Ctrl+W**.

### Close All Drawing

Closes all opened drawing windows and removes the corresponding drawings from the list of drawings in memory. If you cancel any of the save dialogues, the process is canceled and no further drawing windows are closed.

The keyboard shortcut for this function is **Ctrl+Shift+W**.

### Recently saved

The **Recently saved** menu allows you to load recently saved files.

### Export

The items in the export submenu allow you to save the active drawing in several formats for use with other applications. The **Export** menu has three submenus. **Export current drawing** comprises export filters for the active drawing only. All these filters are available through the first menu entry **Export current drawing (any type)**, too, where you can choose the desired export format via a drop-down box in the file dialog.

**Export all drawings (single file each)** provides the same set of filters, but there they are applied to all drawings automatically. This results in one exported file per drawing, these files are stored in the same location and with the same name as the respective drawing files, but with a different extension.

**Export all drawings (merged file)** comprises export filters that are able to merge all drawings into one file. Since this feature must be supported by the format of the exported file, the set of export filters in this submenu is restricted.

The export formats included in the basic Renew distribution are listed as follows:

**PDF** This function produces a PDF document that contains the current drawing. A file with the default extension of **.pdf** is generated.

The “Renew FreeHEP Export” plugin provides the `de.renew.io.export.pageSize` and `de.renew.io.export.pageOrientation` configuration properties, which influence the page layout of generated PDF files. Possible values for page size are: *A3*, *A4*, *A5*, *A6*, *International*, *Letter*, *Legal*, *Executive*, *Ledger* and *BoundingBox*. Possible values for orientation are: *portrait* and *landscape*.

The properties default to *BoundingBox* for page size and *portrait* for orientation. However, orientation does not apply, if page size is set to *BoundingBox*.

The keyboard shortcut for this function is **Ctrl+Shift+P**.



**EPS** If you want to include net drawings into written material, you can use an Encapsulated Postscript (EPS) file. The EPS file can be used to insert graphics into other documents, e.g. in LaTeX, LibreOffice, Microsoft Office, and others. EPS files are not of a fixed page size. Instead, their bounding box matches exactly the dimensions of the drawing.

The keyboard shortcut for this function is **Ctrl+E**.

The EPS and PDF export feature relies on the VectorGraphics package of the FreeHEP project (see <https://java.freehep.org>). The “Renew FreeHEP Export” plugin provides a property for the configuration of the font handling (`de.renew.io.export.epsFontHandling`). It can be set to **Embed**, **Shapes** or **None**. The **Shapes** option is the default as it produces the most similar output with respect to screen display. However, the generated files can become rather large. The **None** option comes close to the old Renew export behavior without any font information included. The **Embed** option should be the best (theoretically), but it often produces unreadable results.

The background of drawings exported to eps can also be set to transparent by setting the property `de.renew.io.export.eps-transparency` to true. Included images with transparency (alpha) always get a white background. If you want to include transparent images, we recommend to use the PDF export.

**PNG** This function produces a PNG image that contains the current drawing. A file with the default extension of `.png` is generated. This export format differs from the previously mentioned formats since it is pixel-oriented instead of vector-based. The generated image has a fixed resolution that cannot be scaled without loss of information. The background of exported PNG images is transparent. The PNG export is based on the FreeHEP library.

The keyboard shortcut for this function is **Ctrl+9**.

**XML** There are several export formats that use XML. We provide experimental PNML support since Renew 2.0. PNML stands for Petri net Markup language and has been presented at the ICATPN’2003 in [2]. With Renew 2.2, the SVG format has been added. With Renew 2.4, the support for the experimental XRN format provided in previous releases has been discontinued.

**PNML-P/T-Net** This format saves the drawing as a P/T-net, compatible with the PNML standard type definition in version 2009 from <http://www.pnml.org>. Note that all drawing elements which are not needed to describe the P/T-net are omitted.

**PNML-P/T-Net with Renew-specific inscriptions** This format saves the drawing as PNML-P/T-Net but stores the Renew-specific inscriptions using the toolspecific extension of PNML.

**PNML-RefNet** This format saves the drawing as a Renew reference net. Graphical figures without semantic meaning (e.g. those figures produced by the drawing tool bar) are omitted. The underlying PNML type definition is experimental, it may be subject to changes without notice.

Please note that the PNML standard allows multiple nets to be contained within one file.

**SVG** This format exports the complete graphical information of a drawing into an SVG image file which can be displayed by many modern web browsers. Petri net semantics are not retained. The SVG export is based on the FreeHEP library.

**Woflan** Woflan (see [7]) is a Workflow Analysis tool that checks if a Petri net conforms to some restrictions that make sense for Workflows. As Woflan only handles single, non-colored Petri nets without synchronizations, only the structure of the active window’s net is exported.



Still, if you have the Woflan tool, it makes sense to check Renew workflow models for severe modeling errors in their structure.

For the time being, the initial place of the workflow net must carry the name `pinit`. Otherwise, a place with this name (but without any connected arcs) will be generated in the exported net.

**Shadow Net System** A *shadow net system* can comprise one or more nets which can be used by the non-graphical simulator (see section 4.5), the net loader or other tools. Only the semantic information is contained in the shadows, but not the visual appearance. Note: Only the original drawings can be exported, not the simulated drawings.

The current formalism (see section 4.3.6) and the configuration of simulation traces for individual net elements (see section 4.3.5) will be stored within the shadow net system.

**As merged file** A shadow net system that contains all nets needed for a simulation can be generated by the `N to 1` entry in the **Export all drawings (merged file)** menu. Before exporting a collection of nets to the shadow simulator, it is recommended to do a syntax check on the net. Although any syntax errors in the nets will be detected before the start of a non-graphical simulation, fixing these errors requires the editor.

The current formalism (see section 4.3.6) and the configuration of simulation traces for individual net elements (see section 4.3.5) will be stored within the shadow net system.

**As single file each** These files are well suited for the net loading mechanism described in subsection 2.6.6.

The command does not require any additional interaction, all file names are derived from the corresponding drawing files. If a drawing has not been assigned a file name, it is skipped during the export.

## Import

The items in the import menu allow you to load drawings from different file formats.

**import (any type)** The first entry of the Menu combines all import filters into one dialog where you can choose the desired format from a drop-down box. For window managers where this drop-down box is not available, the separate menu entries are still available.

**XML** Analogous to the export features described in subsection 4.3.1, Renew provides the PNML format as import filters.

**PNML** Tries to import a file in PNML format. The filter automatically guesses the net type used in the PNML file. It tries to extract as much graphical and semantic information as possible from the file.

**Shadow Net System** Lets you import a previously exported (or automatically generated) shadow net system (see above).

Since a shadow net system does not contain any graphical information, the places, transitions, arcs, and inscriptions are located in a rather unreadable manner. Thus, this function only makes sense for shadow net systems automatically generated by other tools. After importing, it is of course also possible to edit all nodes and inscriptions in a normal fashion. An automatic graph layout function that can ease the task of making an imported net readable is described in Subsection 4.3.3.



## Print...

The print menu invokes a platform dependent print dialog and lets you make hardcopies of the active drawing. Using the Java standard print system, though, the quality of the printer output is usually very poor. In that case, we encourage to use the EPS or PDF export instead and print with an external tool.

The keyboard shortcut for this function is **Ctrl+P**.

## Exit

Tells Renew to terminate. All drawings are closed as if you closed them manually, which means that now Renew asks you about saving changed drawings (see Subsection 4.3.1).

Due to the introduction of the plug-in system, other plug-ins might still be active when the editor is closed. With respect to the simulator plug-in, the editor asks you for confirmation to terminate a running simulation (if there is any). If you choose **No**, then the non-graphical simulation of Renew will continue.

To enforce that the whole plug-in system is shut down when you close the editor, you can configure the property `de.renew.gui.shutdownOnClose` (see Subsection 2.7.3 for details).

## 4.3.2 Edit

The Edit menu contains functions to insert, remove and group figures and to change a figure's Z-order. Details can be found in the following sections.

### Undo, Redo

Up to ten modifications to each drawing can be undone step by step. The effect of an undo can be undone by the redo command. The keyboard shortcut for undo is **Ctrl+Z** and for redo it is **Ctrl+Y**.

### Cut, Copy, Paste

This function group offers the typical clipboard interactions. Cut and Copy relate to the current selection in the active drawing window (see Section 4.1). Thus, these functions are only available if there is a current selection.

**Cut** puts all selected figures into the clipboard and removes them from the drawing. The keyboard shortcut for Cut is **Ctrl+X**.

**Copy** puts all selected figures into the clipboard, but they also remain in the drawing. The keyboard shortcut for Copy is **Ctrl+C**.

**Paste** inserts the current clipboard contents into the active drawing. The upper left corner of the object or group of objects is placed at the coordinates of the last mouse click. The keyboard shortcut for Paste is **Ctrl+V**.

Note that due to restrictions of Java, Renew's clipboard does not interact with your operating system's clipboard.

The current selection is automatically extended to include all referenced figures before copying to the clipboard. If for example you select an arc inscription and invoke copy and then paste, the arc, the start figure, and the end figure of the arc will also be copied. This is sometimes not what you intended to do, but you can easily move the copied arc inscription to the original arc (see Section 4.2.2) and remove the other duplicated figures. Of course, cut only removes the figures which were originally selected.



The better alternative to copy inscriptions is to mark and copy the text of the inscription when you are in text edit mode (**Ctrl+C**, unfortunately, this does not work on all Unix platforms). Then, create a new inscription by right-clicking the net element. Edit the new inscription by right-clicking it and paste the copied text by pressing **Ctrl+V**.



## Duplicate

Duplicate works like Copy followed by Paste (see previous Section), where the paste coordinates are not depending on the last mouse click, but are just a small offset to the right and down from the position of the original selection.

The keyboard shortcut for Duplicate is **Ctrl+D**.

## Delete

Removes the selected figures from the active drawing. Note that if a figure is removed, all its connected text figures and connection figures are also deleted.

The keyboard shortcut for Delete is the backspace and/or the delete key (depending on the platform).

## Search, Search & Replace

**Search** looks for a match or substring-match of an user given search string in all textfields of all loaded nets. Search is case sensitive. After an occurrence of the search string is found the next can be found by pressing the **search** button again. Changes on the search string start a new search. The keyboard shortcut for this function is **Ctrl+F**.

**Search & Replace** gives the opportunity to replace any found search string with a replace string. Each replacement is prompted and has to be confirmed by the user. Changes on the replace string start a new search. The keyboard shortcut for this function is **Ctrl+G**.

The search window allows you to select, whether the search should be case sensitive and whether the search shall include all drawings or only the active one.

## Group, Ungroup

You can create a group of all currently selected figures in the active drawing. A group is actually a new figure, which consists of all the selected figures. You can even group a single figure, which does not really make sense unless you want to prevent resizing of this figure. From now on, the figures inside the group can only be moved, deleted, etc. together, until you “ungroup” the group of figures again. To release a group, one or more groups have to be selected. Then, select the Ungroup menu, and all group participants are single figures again (which are all selected).

## Select All

Commands that allow selection or deselection of large sets of nodes allow the user to select groups of logically related net elements together. For selecting locally related net elements or individual net elements see Subsection 4.2.1.

Using the select all command, all figures of a drawing are selected. This is useful when you want to move all the net elements to a different place. This command works even for figures that are located off-screen. The keyboard shortcut for this function is **Ctrl+A**.

## Invert Selection

Inverts the selection of the drawing: All selected net elements will be removed from the selection, whereas all the other net elements will be selected.

## Select

This menu hierarchy is used to select all nodes of a certain type. E.g., it offers the possibility to select all transitions, or all arcs, or all inscriptions that are attached to places.

This command comes in handy when you want to set attributes like color or font size for all figures of a certain type.



### Add To Selection

This command is similar to the select command, but it does not clear the selection before it selects the net elements, thereby achieving a union of the selection sets.

This command is especially useful when you want to select a combination of net elements that is naturally covered by the selection command itself. E.g., you can select all transitions and then add all inscriptions of transitions to the selection.

### Remove From Selection

This command is the opposite of the add-to-selection command. It removes certain figures from the selection, but leaves the selection state of the remaining figures unchanged.

This command can be used to select all figures, but *not* the transitions or *not* the arcs.

### Restrict Selection

Sometimes you want to select a certain type of net elements inside a certain area. In this case, the restrict command allows you to select the entire area as described in Subsection 4.2.1, but to restrict the selection to a certain type of figures afterward.

The remove-from-selection command can be used instead of this command, if you want to specify the figures to drop from the selection instead of the figures to keep in the selection.

## 4.3.3 Layout

The **Layout** menu allows to snap figures to a grid, to align a figure's position according to other figures, to change the Z-order of figures and to layout graphs automatically.

### Toggle Snap to Grid

Selecting this menu toggles the Snap to Grid mode of Renew. This grid is not absolute referring to the page, but means that when the grid is active, figures can only be placed to grid positions and moved by certain offsets. Because the editor considers offsets while moving (not absolute coordinates), figures should be aligned first (see below) and then moved in grid mode.

The grid density now customizable. Set the option `ch.ifa.draw.grid.size` to the desired value in your preferences. Default is 5 pixel. This preference is dynamic, i.e. can be set at runtime. Effects take place during next execution of command. The grid can also be set as the default behavior for drawing views. Set the `ch.ifa.draw.grid.default` property to *true* (default *false*).

### Align

The commands in this menu align the selected figures. The figure selected first is the reference figure which determines the coordinates for all others.

**Lefts, Centers, Rights.** These commands align the figure's *x*-coordinates, i.e. they move them horizontally. **Lefts** sets the left-hand side of all selected figures to the *x*-coordinate of the left side of the figure that was selected first, **Rights** does the same for the right-hand side. **Centers** takes into account the width of each figure and places all figures so that their *x*-center is below the reference figure's *x*-center.

The keyboard shortcut for aligning middles is `Ctrl+\` (the backslash character).



The shortcut works only on an English keyboard layout, where the keys for the shortcut can be typed directly, i.e. without additional modifiers like Shift.



**Tops, Middles, Bottoms.** These commands work exactly like the commands in the previous paragraph, except that the *y*-coordinate is changed. Thus, figures are moved vertically in order to be aligned with their tops, middles, or bottoms.

The keyboard shortcut for aligning middles is **Ctrl+Shift+-** (the minus sign).

### Spread

The items of this menu spread the selected figures equidistantly. The two outermost figures that are selected stay at their previous location while all other selected figures are repositioned. The order of the figures (left-to-right, top-to-bottom or diagonal) remains unchanged. To use the spread commands, you must have selected at least three figures.

**Lefts, Centers, Rights, Distances.** Here we spread the figures by modifying their *x*-coordinates. The *y*-coordinate remains unchanged. **Lefts** arranges the figures in a way that the *x*-coordinates of their left borders are distributed equally. **Rights** does the same with respect to the figure's right borders, **Centers** with respect to each figure's center point. **Distances** arranges the figures in a way that the space in between of each pair of neighboring figures will have the same width. The differences between the four commands will only be visible when figures of different sizes are selected.

**Tops, Middles, Bottoms, Distances.** These functions work exactly like the functions in the previous paragraph, except that the *y*-coordinate is changed. Thus, the figures are moved vertically to equal the distances of their tops, middles, bottoms or borders.

**Diagonal.** This command spreads the figures in both directions, horizontally and vertically. All figures are treated with respect to their center point. First of all, a virtual line is drawn from the outermost figure (in relation to the center of the bounding box of all selected figures) to the figure most apart from the outermost one. Afterward, all other figures are moved onto that line with equal distances between their center points. The order of the figures on the line is determined by the order of the orthogonal projections of their original location onto the virtual line.

### Send to Back, Bring to Front

The figures in a drawing have a so-called Z-order that determines the sequence in which the figures are drawn. If a figure is drawn early, other figures may cover it partially or totally.

To change the Z-order of figures, the functions **Send to Back** and **Bring to Front** are available. **Send to Back** puts the selected figure(s) at the beginning of the figure list and **Bring to Front** puts it/them at the end, with the result explained above.



Sometimes, certain figures can not be reached to select and modify them. Using these functions it is possible to temporarily move the covering figure to the back, select the desired figures, and move the figure to the front again. Another option in cases like this one is to use Area Selection (see Section 4.2.1).

### Figure size

These two commands set the size of figures. The function **copy within selection** sets the size of all selected figures to the size of the figure that was selected first. The command **reset to default** resets the selected figure's sizes to their figure type specific default. As there are defaults specified only for net element figures, the command will not have any effect on ordinary drawing figures.



## Automatic Net Layout...

Especially for automatically generated nets, it is nice to have an automatic layout of the net graph, so that one gets at least a rough overview of the structure of the net.

The automatic layout provides an option *Random*, which uses a simulated annealing algorithm to automatically control the parameters of the automatic layout.

This menu entry opens a window to control the automatic net layout. The algorithm can be stopped and started with the respective buttons. While this mode is active, the nodes of the net are moved according to certain rules that are to some extent inspired by physical forces acting on a mesh of springs.

- Arcs have a certain optimal length that is dependent on the size of the adjacent nodes. They will act as springs.
- Arcs feel a torque whenever they are not horizontally or vertically oriented. The torque works toward these optimal positions.
- Nodes feel a repulsive force from each other until a certain distance is reached where this force disappears entirely.
- Nodes feel friction, i.e., the motion that was caused by the forces mentioned before continually slows down unless the force is still applied and compensates the friction.
- Nodes that would move out of the upper or left border are pushed back into the viewable area of the drawing.

These rules will not produce the nicest net graph in many cases, but they can ease the early stages of the layout considerably. They might also be used to maintain a layout during early prototyping phases when the structure of a net changes constantly.

In order to improve the layout of the graph, a special window pops up that allows you to control some parameters of the physical model using sliders. The first slider controls the length of the springs. Some diagrams tend to clump together too much, which might be a reason to raise this value. On the other hand, the spring might be too rigid, not allowing some springs to stretch to their optimal length. In that case, you can control the rigidity of the spring with the second slider.

The repelling force acts only up to a certain distance. By default, the force is quite far reaching and establishes a nice global spreading. But you may want to reduce this force's maximum distance in order to exclude only overlapping nodes. In that case, it may also be good idea to increase the repulsion strength.

The torque strength controls whether the arcs are supposed to be very strictly horizontal or vertical. Initially, this force might actually inhibit the progress toward optimal layout, but in the end it helps to get a nice net. Try to vary this slider's position during the layout for optimal results.

Lastly, the friction slider may be lowered, so that the motion is faster overall. Use this slider with care, because the layout algorithm may become unstable for very low friction values and convergence to an equilibrium might actually slow down due to oscillations. The optimal value depends heavily on the topology of the net. If you feel that you cannot set some force's strength high enough, consider lowering the other forces, and also lowering the friction a little.



Even while the graph is changed automatically, you can still grab a node with the selection tool and move it to a desired position. Of course, it might fall back into the old position due to the acting forces, but your action might establish a topologically different situation where the forces act toward a different equilibrium. This is especially useful when you have selected high torque and rigid springs, but low or no repulsion.

After you are satisfied with the graph, switch off the layout mode. If you add or remove nodes or arcs during the layout procedure, you have to restart the net layout algorithm,



before these changes affect the layout algorithm. Note that the start of a layout procedure always affects the current drawing, not the drawing that was previously used for layout.

## Location

Using this menu you can declare the currently selected figures as either fixed or dynamic. Dynamic nodes participate in the automatic layout as usual, which is the default. On the other hand, fixed nodes still exert forces upon other nodes, but they are rigidly glued to their position and move only if the user moves them.

By fixing the location of some nodes, you can select a preferred direction or specify the basic shape of the net while leaving the details to the layout algorithm.

### 4.3.4 Attributes

This menu helps you to change a figure's attributes after its creation. If several figures are selected, the attribute is changed for all figures that support that attribute. If you try to change an attribute that some selected figures do not support (e.g. font size for rectangles), nothing is changed for that figures, but the change is still applied to the other figures.

Since version 4.0 many of the following attributes can also be changed via the attribute buttons in the top right corner of each drawing panel.



Figure 4.4: The Attribute bar.



There is a known issue that the attribute buttons at the upper right corner can act a little bit weird sometimes.

## Fill Color



The fill color attribute determines the color of the inner area of a figure. All figures but the line-based figures like connection, arc, etc. offer this attribute. The values for this attribute could be any RGB-color, but the user interface only offers 14 predefined colors from which you can choose. The default fill color is Aquamarine except for text figures, where it is None.

When you choose **other...** at the end of the list of colors, you get a full-featured color chooser dialog that provides multiple ways to define any color. There are four buttons at the bottom of the dialog:

**Apply** applies the currently chosen color to selected figures.

**Update** chooses the color of a selected figure and makes it the current color in the dialog.

**OK** closes the dialog and applies the currently chosen color to selected figures.

**Cancel** closes the dialog.



The dialog can be used to copy color attributes between figures by a sequence of **Update** and **Apply** actions. Similar dialogues are provided for other attributes like pen color, text color, font and font size.



## Opaqueness



The opaqueness attribute determines the transparency of the inner area of a figure, of the pen color or of the font. Each attribute **Fill Color**, **Pen Color** and **Font** has its own opaqueness menu that is located right below each menu entry. The visibility of each item can be set in values ranging from 0% (invisible) to 100% (opaque).

Via the menu bar the opaqueness can only be changed in steps of 10% but the attribute button allows for more fine-grained control via a slider.



The transparency attribute is ignored in EPS export. However, transparencies are printed correctly using the Print dialog and in the PDF, SVG and PNG export formats.

## Pen Color



The pen color attribute is used for all lines that are drawn. All figures but the image figure support this attribute. Note that the pen color does not change a text figure's text color (see below), but the color of a rectangle frame that is drawn around the text. Again, choose the desired color from the given list. The default pen color is black, except for text figures, where it is None (i.e. transparent). The **other...** entry at the end of the list of colors opens a full-featured color chooser dialog as described under **Fill color**.

## Visibility

The visibility attribute can be used for all types of figures. This attribute can not be changed via attribute buttons. A figure marked as invisible is still part of the drawing, but it will not be displayed. As it is not visible, it cannot be selected by the mouse any more, but the select commands from the menus **Edit** or **Net** will still include the figure when appropriate. This feature is useful especially in combination with the **Associate highlight** command from the **net** menu. The invisible figure will appear in the instance drawing while it is highlighted.

## Arrow

This attribute is only valid for the connection and the arc figure and offers four possibilities of arrow tip appearance: **None**, **at Start**, **at End**, or **at Both** ends of the line. If the figure is an arc, its semantics are changed accordingly.

## Arrow shape



This attribute is valid only for lines or connection figures. The style of arrow tips can be changed to one of four shapes, which are usually used to mark different semantics of arcs in Renew. But as it is currently not possible to change the arc semantics in accordance to the arrow tip shape, this attribute will not have any effect on arc figures.

## Line Style



Every line possesses a line style, which can be chosen out of the options **normal**, **dotted**, **dashed**, **medium dashed**, **long dashed** or **dash-dotted**. Lines are typically created as solid, normal lines.

It is also possible to define your own line style: After choosing the option **other...**, you can enter any custom line style in a non-modal dialog. The dialog has four buttons **Apply**, **Update**, **OK** and **Cancel** that work similar as in the **Fill color** dialog (see above).

A custom line style consists of a space-separated sequence of numbers. The first number of the sequence determines the length (in pixels) of the first dash. The second number is interpreted as the length of the gap after the first dash. The third number determines the second dash's length, then the next gap's length follows and so on.







### Font Size



Only for text-based figures, select one of the predefined font sizes given in point with this menu. The **other...** entry at the end of the list opens a dialog where you can enter any number as size. The dialog has four buttons **Apply**, **Update**, **OK** and **Cancel** that work similar as in the **Fill color** dialog (see above).

### Font Style



Available font styles (again, only for text-based figures) are Italic and Bold. If you select a style, it is toggled in the selected text figure(s), i.e. added or removed. Thus, you can even combine italic and bold style. To reset the text style to normal, select Plain.

### Text alignment



The direction of text justification can be configured by this attribute. This will affect the alignment of lines in text figures with multiple lines as well as the direction of growth or shrinking when a text changes its width due to a change in its text length. By default, inscriptions and other connected text is centered at the parent figure while other text figures are left-aligned.

### Text Color



The text color attribute is only applicable to text-based figures and sets the color of the text (sic!). This is independent of the pen and fill color. The default text color is (of course) black. The **other...** entry at the end of the list of colors opens a full-featured color chooser dialog as described under **Fill color**.

### Text Type

This attribute is quite nice to debug your reference nets quickly. The text type determines if and what semantic meaning a text figure has for the simulator.

If a text figure is a **Label**, it has no semantic meaning at all. If it is a **Inscription**, it is used for the simulation (see Section 4.2.3: The Inscription Tool). A **Name** text type does not change the simulation, but makes the log more readable (see Section 4.2.3: The Name Tool).



It is quite convenient to “switch off” certain inscriptions by converting them to labels if you suspect them causing some problems. This way, you can easily re-activate them by converting them back to inscriptions.

You might also want to have certain inscriptions appear as transition names during the simulation. You can achieve this by duplicating the inscription figure, dragging the duplicate to the transition (see Section 4.2.2: The Connected Text Tool) and changing the duplicate’s text type to **Name**.

## 4.3.5 Net

This menu offers commands that are useful for nets only. You can semantically modify figures in a drawing, check the active drawing for problems, or configure the graphical simulation feedback for net elements.

### Split transition/place

This command provides a simple way to refine net elements by splitting a single transition or place into two.



If a transition is split the old transition is connected to a newly created place. This place, in turn, is connected to a newly created transition. The inbound arcs of the old transition remain unchanged, the outbound arcs are reconnected to the new transition. Reserve arcs are split into an inbound and an outbound arc, which are handled respectively.

If a place is split it will be extended by a new transition and a new place. The connected arcs are treated in the same manner as described above (outbound arcs are reconnected to the new place).

### Coarsen subnet

This command coarsens place-bounded or transition-bounded subnets. It is only available if a place-bounded or transition-bounded subset of figures is selected within the drawing.

On execution, if the selected subset is place-bounded, all places are merged into one and all transitions are removed. The inscriptions of the removed places are attached to the single remaining place. All arcs entering or leaving the selected subnet are reconnected to this place, too.

If the selected subset is transition-bounded, transitions are merged and places are removed, respectively.

### Trace

This menu and the next two are realized as figure attributes that can be applied to each single net element. However, they must be set before the simulation is started to take effect. They also cannot be applied to figures in net instance drawings.

Sometimes, the simulation log becomes very complex and full. To reduce the amount of information that is logged, the trace flag of net elements can be switched off.

- If a transition's trace flag is switched off, the firings of this transition are not reported in the log window.
- A place's trace flag determines whether the insertion of the initial marking into the place should be logged.
- If an arc's trace flag is switched off, the messages informing about tokens flowing through this arc are omitted.

With the integration of the Log4j framework (see Section 2.6.9), the need for the trace attribute has been reduced. The configuration of Log4j is much more flexible, it allows for multiple log event targets with individual filter criteria while the trace flag globally controls the generation of log events for a net element. A valid reason to still use the trace attribute may be the simulation speed when you want to discard the trace anyway, but Log4j is rather efficient in such a situation, too. Please note that Renew provides a graphical Log4j configuration dialog for simulation traces (see Subsection 4.3.6).

### Marking

This menu controls the default as well as the current choice how the contents of each place is to be displayed during simulation.

There are four ways to display the marking of a place during simulation: Either the marked places are simply highlighted in a different color (**highlight only**), or the number of tokens is shown (**Cardinality**), or the verbose multiset of tokens (**Tokens**) is shown, or each token and its attributes is shown in detail (**expanded Tokens**). This is also the default mode for current marking windows. However, these modes can be switched at drawing time and at simulation time using the **Marking** menu.

The expanded token mode relies on the undocumented feature structure (fs) formalism to display object attributes. Since the fs formalism is not any longer distributed with the base renew distribution, this mode is not available unless you install the FS plug-in.



In Expanded Tokens mode, token objects are shown in a UML-like (Unified Modeling Language) notation. An object is noted by a box containing two so-called compartments.

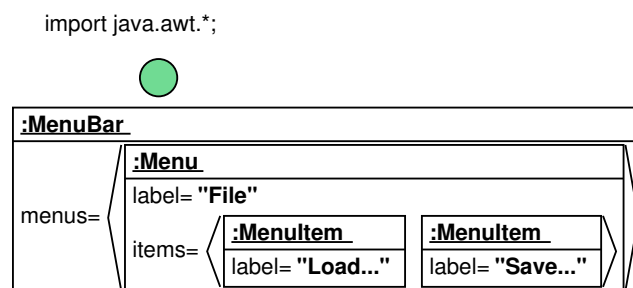


Figure 4.5: An Example of Browsing Token Objects in Expanded Tokens Mode

The first compartment specifies a temporary name of the object (Renew just gives numbers to objects), followed by a colon (:), followed by this object's class name. According to UML, the whole string is underlined to indicate that this is an instance, not the class. The second compartment is only shown if you click the shutter handle, a small yellow rectangle with a cross (plus sign) inside. Otherwise, the available information is indicated by three dots (...) after the class name.

The second compartment contains a list of all attributes of the token object and their values, which are basic types or again objects. Multi-valued attributes (e.g. array values or **Enumerations**) are shown as lists in sharp brackets (this part is not quite UML). After opening the attributes compartment, the handle changes to a horizontal line (minus sign) and lets you close the compartment again if you wish to do so. This way, you can browse the object graph starting at the token object. If the value of an attribute happens to be an object that already appeared in the open part of the object graph, only the temporary name (number) of that object is displayed as the attribute's value. To help you find the original object, you can click on this object number, and all appearances of this object are highlighted by a red frame. To get rid of the highlighting, just click on any of the numbers again.

Figure 4.5 shows an example of a `java.awt.MenuBar` object that is being browsed as an Expanded Token. In the example, the menu bar contains one menu **File** with two menu items of which the first one is **Load...**. The **parent** of the first menu item is again the menu, as you can see by the highlighting. The second menu item is closed.

Renew tries to find attributes of the token object by using Java's reflection mechanism on fields and **get**-methods. Any method without parameters and with a return type which is not **void** is regarded a **get**-method. In some cases, such methods return volatile (changing) results, but are only queried once when the token figure is expanded. This means you should not expect to see changes of a token object while browsing it!

Renew stores for each place the preferred display mode chosen by the **Marking** menu. This means that every new simulation starts with the display mode chosen for each place, and the display mode is also saved to disk. The menu can also be used to change the display mode during run-time. To do this, either the token figure or the place instance has to be selected.

## Breakpoints

Using this attribute, you can request breakpoints for certain places and transitions. These breakpoints will be established immediately after the start of the simulation and have exactly the same effect as a global breakpoint that is set during the simulation. In the net drawing, transitions and places with a set breakpoint attribute are marked by a small red circle in their upper right corner. However, the tag is not shown in instance drawings.



Attributed breakpoints, like breakpoints set during the simulation, will show up in the breakpoint menu while the simulation is running. Please see subsection 4.3.6 for a detailed description of the possible breakpoints. Note that you can set at most one breakpoint for each net element using this menu command.

Attributed breakpoints are established only when the net drawing is loaded in the editor at the moment where the compiled net is passed to the simulation engine. For the initial drawings (that were used to start the simulation) this is usually the case. But if nets are loaded later by the net loader from `.sns` files (see Subsection 2.6.6), no breakpoints are set.

This behavior is due to the fact that the responsibility for the creation of breakpoints lies in the graphical user interface and not in the simulation engine. Since the breakpoint attribute is dropped when exporting shadow net systems (see Subsection 4.3.1), the simulator is not able to establish these breakpoints.

### Set Selection as Icon

This feature allows you to assign icons to your nets. These icons will be displayed during simulation, whenever a place marking is displayed in `token` mode (see subsection 4.3.5) and references an instance of a net with an icon.

Select exactly one figure, which can be of any type, then select the menu **Set Selection as Icon**. If more than one figure was selected, nothing happens, but in the case of a single figure, it is assigned as the net's icon. When the figure is removed, the net does not have a special icon, so that references to this net are again displayed as text. When the figure is or includes a text figure, the string `$ID`, contained anywhere within the text, has a special meaning: During simulation, `$ID` will be replaced by the index number of the referenced net instance.

You can use net icons as in the following example which can be found in the samples folder `icon`. Remember the Santa Claus example from Section 3.7? Imagine you want to visualize the bag and its contents as icons. Figs 4.6 and 4.7 show modified versions of the nets from the Santa Claus example.

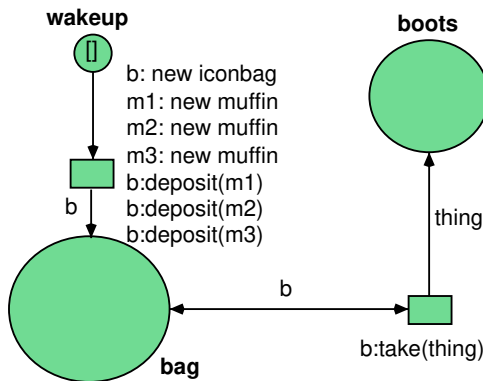


Figure 4.6: The net `iconsanta`

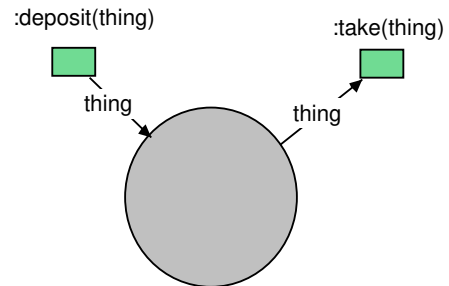


Figure 4.7: The net `iconbag`

Add an icon to the `bag` net by drawing an ellipse, coloring it gray, and drawing a polygon which looks like the closure of the bag. Add a text with the string `BAG $ID` to the drawing. Group together all new figures (**Edit | Group**). This is necessary, since the icon of a net has to be a single figure. Now you can select the group and then the menu **Set Selection as Icon**. Note that when you have to **Ungroup** the icon (e.g. to move one of the included figures individually), this corresponds to removing the group figure. So, after re-grouping the icon, you have to invoke the menu again, or the group figure will not be set as the net's icon.

The next step to make an iconized version of the Santa Claus example is to create a new



net, add an image figure with your favorite sweet (in my case, this is a muffin) and a text figure saying \$ID. Then again group together the image and the text, select this new group, and select the menu **Set Selection as Icon**. Save this net as **muffin**.

Now, you can select the net **iconsanta** and start a new simulation. After performing two steps, the running nets may look like those in Figure 4.8. Note that the reference to the net **bag** is now display as the bag icon with \$ID replaced by the net instance index 1. Without the icon, the token would have been described as **bag[1]**. Also note that the muffins all have different index numbers, so that you can see to which net they refer.

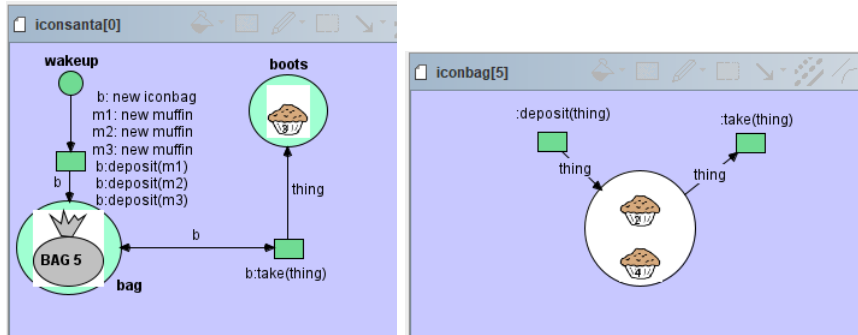


Figure 4.8: The Santa Claus Example with Icons During Simulation.



The background of expanded tokens in instance/simulation drawings is not transparent by default to improve readability. It can be changed to be transparent by setting the property `de.renew.gui.noTokenBackground`.

### Associate Highlight

It is not only possible to select the kind of feedback given for the marking of a place (see Subsection 4.4.1), but also to specify arbitrary graphical elements to be highlighted whenever a place is marked or a transition is firing. Each net element can have at most one highlight figure, but this figure can be any Renew drawing figure like any rectangle, line, text, etc., even a group figure.

You can for example draw a StateChart with Renew's drawing facilities, construct a net which simulates the StateChart's behavior, and associate figures such that during simulation, the StateChart is highlighted accordingly.

The first function one needs for dealing with such highlights is to associate a highlight to a net element such as a place or a transition. When the menu **Associate Highlight** is invoked, exactly two figures have to be selected, of which one has to be a place or a transition.<sup>2</sup> The status line tells you if associating the highlight to the net element was successful, otherwise displays an error message.

Now, during simulation, the associated figure will be highlighted exactly when the net element is highlighted. If the associated figure is invisible, it will be made visible whenever it is highlighted. If the figure is already visible, its color will change as a result of the highlighting.

### Select Highlight(s)

To find the associated highlight figure (see above) to a net element, select the net element and then this menu. If the net element does not have any highlight figure, a corresponding message

<sup>2</sup>It is even possible to associate another net element as a highlight, but this is not recommended, as it can lead to confusion.



appears in the status line. You can also select multiple net elements, and all associated highlight figures of any one net element of the group will be selected.

### **Unassociate Highlight**

Sometimes you also want to get rid of a highlight-association (see above). Then, select one single net element (place or transition) with an associated highlight figure and then invoke this menu. When you associate a net element to a highlight figure, any old association is automatically canceled.

### **Syntax Check**

This menu entry checks the net for syntax errors without starting a simulation run. Of course, most syntax errors are immediately reported after the editing of an inscription, but not all errors are found this way. E.g., multiple uplink inscriptions cannot be detected immediately. You can also invoke a syntax check when you have corrected one error, in order to make sure that no other error remains. It is always a good idea to keep the nets syntactically correct at all times.

### **Layout Check**

This menu entry checks in all loaded drawings whether textfields overlap by more than 50%. Overlap indicates problems in the clear representation. Also, the situation is detected that a second inscription is accidentally assigned to an arc and is hidden because of the overlap.

## **4.3.6 Simulation**

This menu controls the execution or simulation of the net system you created (or loaded). Before a simulation can be started, all necessary nets must be loaded into memory (see subsection 4.3.7). The drawing window containing the net that is to be instantiated initially has to be activated.

Refer to Section 4.4, if you want to learn how to monitor and influence a simulation run that you have started using this menu.

### **Run Simulation**

This function starts or continues a simulation run that continues automatically until you stop the simulation. If you want to enforce starting a new simulation run, use **Terminate Simulation** (see below) first. For most net models, it is almost impossible to follow what's going on in this simulation mode. Its main application is to execute a net system of which you know that it works.

Some syntax checking is done even while you edit the net (see Section 4.2.3: The Inscription Tool), but when you try to run a simulation of your reference nets, the reference net compiler is invoked and may report further errors (see Section 4.6). You have to correct all compiler errors before you can start a simulation run.

The keyboard shortcut for this function is **Ctrl+R**.

### **Simulation Step**

This menu performs the next simulation step in the active simulation run or starts a new simulation run if there is no active simulation.

If a simulation is already running in continuous mode, one more step is executed and then the simulation is paused to be continued in single-step mode. Thus, it is possible to switch between continuous and single-step simulation modes.

The keyboard shortcut for this function is **Ctrl+I**.



## Simulation Net Step

This menu entry performs a series of simulation steps in the active simulation run or starts a new simulation run if there is no active simulation. The simulation is paused when an event in the net instance in the current instance window occurs.

The keyboard shortcut for this function is **Ctrl+Shift+I**.

## Halt Simulation

This menu halts the current simulation run, which has been started with **Run Simulation**, or terminates the search for a possible binding in single step mode. No further simulation steps are made, but you are free to resume the simulation with **Run Simulation** or **Simulation Step**.



There are situations where a net invokes a Java method that does not terminate. In these cases **Renew** cannot succeed in halting the simulation.

The keyboard shortcut for this function is **Ctrl+H**.

On MacOS systems, **Cmd+H** is bound system-wide to hide the application window. Therefore, the shortcut key has been changed to **Shift+Cmd+H**.

## Terminate Simulation

This menu entry stops the current simulation run (if there is any). For certain reasons, the simulator can not know if the simulated net is dead (it could always be re-activated from outside, see Section 3.9), so a simulation only ends when you invoke this command. When you issue another simulation command after this command, a new simulation is automatically started.

All simulation related windows (net instances, current markings, now also possible transition bindings) are now automatically closed when simulation is terminated, since they cannot be used after simulation anyway.

The keyboard shortcut for this function is **Ctrl+T**.

## Configure Simulation...

This dialog allows to change some simulation related configuration options. These options can also be controlled from the command line or the configuration file **.renew.properties** (see section 2.6.1). All options presented in this dialog are evaluated each time a new simulation is started. However, the settings in this dialog are not stored permanently.

The dialog comprises several tabs, each tab groups some configuration options. The buttons at the bottom of the dialog affect all tabs.

**Apply** passes the current settings to the plug-in system, so that the simulator plug-in can interpret them at the next simulation startup.

**Update** refreshes the dialog to display the current settings known to the plug-in system.

Unless you modify some properties concurrently, you can think of this button as a “revert” button, that restores the most recently applied configuration.

**Update from simulation** refreshes the dialog to display the configuration of the running simulation, if there is any. These settings may differ from the current simulator plug-in configuration, so you might want to press **Apply** or **OK** afterward to bring the plug-in configuration back in sync with the settings of the running simulation.

**OK** applies the current configuration (like **Apply** would do) and closes the dialog.

**Close** closes the dialog and discards any setting changes (unless they have been applied before).

The tabs provide the following options:



**Engine** The two options **Sequential mode** and **Multiplicity** configure the concurrency of the simulation engine. The sequential mode is of interest when you work with a timed formalism (see section 3.11) or special arc types (see section 4.3.6). Multiple simulators may enhance the performance on multiprocessor systems. A sequential mode with multiplicity greater than one is not sequential because it uses multiple concurrent sequential simulators.

The settings are equivalent to the `de.renew.simulatorMode` property mentioned in sections 2.6.3 and 2.6.4. Just think of the **Sequential Mode** check box as the sign of the `simulatorMode` value (if you enter a minus sign in the **Multiplicity** field, it is ignored).

The **Class reinit mode** setting equivalents the `de.renew.classReinit` property explained in section 2.6.5. It allows you to reload custom classes during development.

The **Simulation priority** sets the priority of each thread the simulation spawns. Higher values allow for faster simulations but might result in reduced GUI responsiveness. The default value of 5 is considered a good tradeoff between speed and gui response time.

**Remote Access** The options provided by this tab find their equivalents in the remote properties which are explained in section 2.6.8. When you check **Enable remote access**, the simulation will be published over Java RMI to allow remote inspection and simulation control (this feature needs a running RMI registry to work). To distinguish multiple simulations on the same registry, you can assign a **Public name** to the simulation. Plug-In developers might be interested in the possibility to replace the remote **Server class** by a custom implementation. A custom RMI **Socket factory** can only be supplied at startup, therefore this property cannot be changed here.

To observe the simulation from a remote editor, use the the **Remote server** command explained in section 4.3.6.

**Net path** This tab allows the manipulation of the `de.renew.netPath` property used by the net loader (see section 2.6.6). On the left, you have a list of path entries, one directory per line. The net loader searches the directories in order from top to bottom. In the list, you can select one or more entries to manipulate. On the right, there are five buttons, most of which affect the selected set of entries.

**Add...** opens a dialog where you can enter a new path entry. The directory should be entered in os-specific syntax.

If you want to specify a directory relative to the classpath, check the appropriate box and make sure that the path does *not* start with a slash, backslash, drive letter or what else declares a path absolute at your operating system.

**Edit...** opens a dialog similar to the **Add...** dialog for each selected path entry.

**Move up** moves all selected entries one line above the first selected entry (or to the top of the list, if the topmost entry was included in the selection).

**Move down** moves all selected entries one line below the last selected entry (or to the end of the list, if the bottom-most entry was included in the selection).

**Delete** removes all selected entries from the list.

**Logging** This tab configures the simulation log traces (see Menu entry “Show simulation trace...” below). In contrast to other tabs, changes to the settings on this tab take effect immediately.

It is possible to create additional loggers that focus on net-, transition- or place-specific parts of the simulation trace. A click with the right mouse button on the top-level entry of the logger tree opens a context menu where additional loggers can be added. The logger name serves as filter criterion.



Each logger can be configured to send its data to one or more appenders. Depending on the kind of appender, the filtered simulation trace can go to the console, a file or a trace window (**GuiAppender**). Each appender can be configured with various options. For example the buffer size (the number of viewable simulation steps) of the **GuiAppender** can be adjusted to your needs.

The textfield **Layout** is used to customize logger output using `log4j PatternLayout`.

## Breakpoints

You can set breakpoints to stop the simulation at a predefined point of time, which is especially helpful for debugging purposes, where the simulation might have to run for extended periods of time, before an interesting situation arises.

The breakpoint menu consists of two sections. The first allows you to set and clear breakpoints and the second allows you to view all breakpoints currently set in the simulation.

A breakpoint will stop the search for enabled bindings when running a simulations. However, the execution of those transitions that are already firing continues. This is especially important if a breakpoint is attached to a transition: The transition might still run to completion while the breakpoint is reported.

That means that you will often want to attach a breakpoint to an input place of a transition, if you want to inspect the state of the net before a certain transition fires. You cannot currently detect a change of enabledness directly.

**Set Breakpoint at Selection.** Before setting a breakpoint you must select a place or transition or a group thereof within a net instance window. You can set a breakpoint either locally or globally. A local breakpoint will affect exactly the chosen net instance and will not cause a simulation stop if other net instances change. A global breakpoint automatically applies to all net instances, even those that will be created after the breakpoint is established.

There are a number of different breakpoint types:

- **Default.** This is a convenience type that is equivalent to a breakpoint on start of firing for transitions and on change of marking for places. You can use it if you want to set a breakpoint to a place and a transition simultaneously.
- **Firing starts.** This breakpoint is triggered whenever the transition starts firing. The breakpoint happens just after all input tokens have been removed from their places and the transition is about to execute its actions.
- **Firing completes.** Unlike the previous item, the breakpoint occurs at the end of a transition's firing. This is especially useful in the case of net stubs, where you want to inspect the result of a stub call.
- **Marking changes.** Any change of the state of a place is detected here, even if the change is simply due to a test arc.
- **Marking changes, ignoring test arcs.** Here it is required that tokens are actually moved and not merely tested.
- **+1 token.** Only a token deposit triggers this breakpoint.
- **-1 token.** A token removal must occur before this breakpoint is activated.
- **Test status changes.** Normal arcs do not trigger this breakpoint, but test arcs do.

Multiple breakpoint types may be set for a single net element using this menu.

**Clear Breakpoint at Selection.** A breakpoint is not automatically cleared after it was invoked. Instead, you must clear breakpoints explicitly. Having selected the net element that contains a breakpoint, you can either clear all local breakpoints or all global breakpoints.



**Clear All Breakpoints in Current Simulation.** This command will get rid of all breakpoints that were ever set. This is useful if you have reached a certain desired situation and want to continue the simulation normally. Alternatively, you might want to clear all breakpoints that were configured using the attribute menu, if you require a completely automatic run once in a while, but not want to lose the information about the standard breakpoints.

**Breakpoint List.** The second part of the menu allows you to view all breakpoints, locate the associated net elements, and possibly reset individual breakpoints.

### **Save simulation state...**

This menu entry saves the current simulation state to a file, so it can be restored later on by the menu command **Load simulation state**. The saved state also includes all net instances currently opened in drawings and all compiled nets. The default extension for Renew simulator state files is `.rst`.

Points to be aware of:

- Saved simulation states will most likely not be compatible between different versions of Renew.
- All custom classes used in the current marking of the net must implement the interface `java.io.Serializable` in a sensible way to obtain a complete state file.

There are also some minor side effects:

- This command halts the simulator, because there must not occur any changes to the current simulation state while it is saved to obtain a consistent state file. You can continue the simulation afterward.
- The binding selection window will be closed, if it is open.

### **Load simulation state...**

This menu entry loads a simulation state from a file saved by the menu command **Save simulation state** before. You will then be able to continue the simulation as usual from the point at which the simulation state was saved.

If all drawings used in the state are loaded, you can use all simulation control facilities as usual. However, it is not necessary to have all used drawings open. If some drawing is missing, the only drawback is that its net instances will not be displayed in instance drawings. As a consequence, you will not be able to use the extended control features described in Section 4.4 for these nets, but the menu commands **Simulation step** and **Run simulation** will still work and trace events will still be logged. This holds even if no drawing used by the saved simulation state is loaded at all.

The mapping from a compiled net contained in the saved state to an open net drawing is done by the net's name. This mapping occurs every time when you try to open an instance drawing for any instance of the net. If you added to or removed from the net drawing any transitions or places since the simulation state was saved, some messages informing you about the problem and its consequences are printed to the application log. An instance drawing will still be opened, but it will not necessarily display the same structure that the compiled net uses.

Further points to be aware of:

- If you load a simulation state, any running simulation will be terminated and all related windows are closed.
- If the class reinit mode is selected (see Subsection 2.6.5), custom classes will be reloaded while restoring the simulation state.
- All custom classes used in the saved simulation state must be available when restoring the state.



### Remote server...

Using this menu entry, you can list all net instances of a Renew simulation server. To be able to do this, a simulator must be running with remote access enabled as described in section 2.6.8.

The dialog comprises two parts: The upper buttons switch between remote simulations, the lower part shows a list of net instances. Initially, the list shows net instances of the local simulation (if there is a running simulation).

The **Connect...** button displays another dialog which allows you to connect to a remote simulation server. You must specify the host on which the **Server** is running. The server **Name** can be left at the default value unless you specified the `de.renew.remote.publicName` property on the server side.

If the connection has been established, the drop-down box at the top of the **Remote Renew servers** dialog includes the remote simulation and the list of net instances is updated. You can switch between servers by selecting them in the drop-down box. The connection stays alive until you press the **Disconnect** button, or either Renew application (local or remote) terminates.

In the net instance list, you can select a net instance and open it by double-click or by pressing the **Open** button. The title of the net instance window shows that it is the instance of another server. You can use nearly all the interaction features of local net instance drawings. All your modifications are executed on the server. Like local simulation windows, events from the remote simulation ensure that the drawings will be up-to-date at every time.



The editor is not able to display two net instances with the same name and id. It will bring the existing net instance window to front when you select a net instance with the same name and id from a different simulation. To see the other net instance, close the existing net instance window.

### Show simulation trace...

This menu command opens a window that shows the trace of the current simulation. In previous Renew releases, the trace has always been printed to the console, now you can closely inspect the trace inside the editor. The shortcut for this command is **Ctrl+L**.

By default, the drop-down list on top of the window provides one simulation trace that covers the last 20 simulation steps. You can configure additional traces of different length that focus on specific net instances, places or transitions using the **Logging** tab of the **Configure simulation** dialog (see above).

A double left mouse button click on a simulation trace entry opens a window that displays the whole message, using multiple lines if appropriate. A right mouse button click opens a context menu that allows you to display the net template or instance that was involved in the simulation step. It is also possible to select the individual place or transition in the net template or instance.



The mouse actions to inspect a trace entry are not available before you have selected any line of the simulation step it belongs to.

### Formalisms

This submenu configures the current formalism used during compilation and simulation. Please note that a running simulation will always stay with the formalism it has been started with. To apply the chosen formalism to the simulation, you have to terminate it and start a new one.

The entries of this menu depend on the set of plug-ins currently installed. The basic renew distribution includes four formalisms, represented by their compilers:



**Timed Java Compiler** represents the same formalism as the **Java Net Compiler**, but with additional time annotations as explained in section 3.11. Nets compiled by this compiler must be executed in a sequential simulation.

**Bool Net Compiler** compiles nets according to the formalism presented in [13]. A bool net is a restricted colored net with exactly one color  $bool := \{0, 1\}$  (can also be represented as  $\{\text{false}, \text{true}\}$ ). It accepts one of the propositional logic operators **and**, **or** and **xor** as transition guard inscriptions.

**Java Net Compiler** encapsulates the reference net formalism with Java inscriptions as described in chapter 3. However, this compiler does not accept time annotations.

**P/T Net Compiler** compiles the net as a simple place-transition net. It accepts integer numbers as initial markings and arc weights. Capacities are not supported.

The PTChannel plug-in for simulating P/T nets with synchronous channels allows for selection between another two options. It allows for transition inscriptions, with the keyword for the downlink being **this** and also for transition channels to use variables and integers.

**Single P/T Net with Channel Compiler** can be used within a sole net instance.

**P/T Net in Net Compiler** can be used for two or more net instances. In order to have a system net for communication, hyperparameters have to be defined.

#### Show sequential-only arcs

This option is available only when the **Java Net Compiler** is chosen as current formalism. Selecting this option adds another toolbar to the editor. This toolbar comprises two additional arc types (see section 4.2.3) which are allowed in sequential simulations only. Please note that this option is automatically enabled (although the menu entry is not visible) when you choose the **Timed Java Compiler** as formalism.

For your convenience, the sequential simulation mode (see sections 4.3.6 and 2.6.4) is activated each time you check the box or choose the **Timed Java Compiler**. However, the engine is not switched back to concurrent mode when you uncheck the box or change to another formalism.

### 4.3.7 Windows

This menu contains a list of all opened drawings. Clicking on an entry closes the drawing.



In previous versions of Renew, the **Windows** menu contained a list of opened windows, including drawings, tool dialogs and attribute dialogs. Clicking on an entry in the **Windows** menu raised the corresponding window. This functionality had to be temporarily removed in Renew 4.0 with the GUI update. It will be restored in a future release.

### 4.3.8 System

This menu offers different styles that can be used to change the optical style of Renew.

### 4.3.9 UI Scale

This function lets you scale the entire UI by a specified factor. On click, a dialog will open that lets you set a permanently persisted percentage value. To apply the changes, a restart is required. You will be prompted for it. If you choose to restart, the usual **Save Drawing...** functionality will automatically be executed before.



Be aware that fractional scaling values may not work on Linux. Try e.g. "200" instead.



### 4.3.10 Additional Top-Level Menus

The menu manager allows for the registration of a menu item by the plugins under any top-level menu. Additionally, plugins may use a new top-level name. Typical candidates are *Plugins*, *Tools* and *Application*

The optional plugin *GuiPrompt* offers its command under the *Plugins* menu. The *Net-Components* plugin and the optional plugins<sup>3</sup> *NetDiff* and *Lola2* reside under the *Tools* menu. Since version 2.3 it is also possible to determine the position of the menu item within the menu. The *Navigator* plugin extends the *File* menu.

## 4.4 Net Simulations

During simulation, there may be textual and graphical feedback. The Log4j framework receives simulation events and can log them alternatively to the console, a file, the trace window, etc. In Subsection 4.3.6, the graphical configuration dialog for Log4j is explained. In a trace of log events, you can see exactly which transitions fired and which tokens were consumed and produced. Alternatively, you can view the state of the various net instances graphically and you can influence the simulation run. The following sections describe the means to monitor and control the simulation.

### 4.4.1 Net Instance Windows

The graphical feed-back consists of special windows, which contain instances of your reference nets. When a simulation run is started, the first instance of the main reference net that is generated is displayed in such a net instance window. As in the simulation log, the name of a net instance (and thus of its window) is composed of the net's name together with a numbering in square brackets, e.g. `myNet[1]`. Net instance windows can also be recognized by their special background color (something bluish/purple), so they cannot be confused with the windows where the nets are edited. In a net instance window, you cannot edit the net, you cannot even select net elements. The net is in a “background layer”, and only simulation relevant objects are selectable, like current markings of places and transition instances. Places in net instance windows are annotated with the number of tokens they contain (if any). If you double-click on a marking the containing place will be selected. If you right-click on such a marking, the marking will switch between the number of tokens and the tokens in a string representation. If you right-click on the containing place, another window appears, containing detailed information about the tokens.

You can display the contents of the current marking directly inside the net instance window. This is extremely useful when a place contains only few tokens (or even only one). This also helps to control the number of windows, which could become very large using Renew. To switch between the simple (cardinality of the multiset) and the token display of a place marking, just right-click it. The expanded display behaves exactly like the contents of a current marking window, which is described in the following section. Tokens in markings are always displayed with a white, opaque background. This increases the readability of markings.

### 4.4.2 Current Marking Windows

A current marking window shows the name of the corresponding place (net instance name dot place name, e.g. `myNet[1].myPlace`) in its window title bar. If the token list does not fit into the current marking window, the scroll bars can be used. For each different token value in the multiset, a current marking window shows the multiplicity (if different from one) and

---

<sup>3</sup>All mentioned optional plugins are not part of the release of Renew. They are provided separately.



the value itself. The Expanded Tokens mode described in Subsection 4.3.5 is now the default mode for current marking windows (if the FS plug-in is installed).

There is a special function to gain access to other net instances. If a token's value is or contains a net instance, a blue frame appears around the name of the net instance. If you click inside that frame, a new net instance window for that net instance is opened or the corresponding net instance window is activated, if it already existed. This also works for net references contained within a tuple, or even within a nested tuple. Using the Expanded Tokens mode, this also works for net references contained within a list or inside any other Java object



You can open a net instance window, double click all places you want to “watch” and close the net instance window again. This helps to focus on the state information you really want to see.

### 4.4.3 Simulation Control

In a concurrent system, many transitions can be activated at once. Normally, the simulation engine decides which of these transitions actually fires when the next simulation step is executed. For debugging and testing, it can be very convenient for you to take care of this decision. Of course, this only makes sense when the simulation is performed step by step (see below).

Interactive simulation is possible. You can force a specific enabled transition to fire in two ways:

- Right-click the transition. Here, the simulation engine still decides nondeterministically about the variable bindings.
- Double-click the transition. Then, the so-called binding selection window is shown and switched to the transition you double-clicked. The title of the window says “*transition-name*’s possible bindings”, where *transition-name* is the full name (name of the net instance-dot-transition-name) of the transition.

In the top part of the window a single binding is described. Each transition instance that participates in this binding is shown on a single line, listing those variables that are already bound. See Section 3.7 for an explanation why multiple transition instances might participate in a single firing. At the bottom of the window there is a list of all possible bindings, where each binding is displayed in a single row.

When you press the **Fire** button, the binding of the entry which is currently selected will be used in the firing. This window should be automatically updated whenever the net’s marking changes. Use the **Update** button, if the automatic update fails, and make sure to report this as a bug. **Close** hides the transition binding window.

If the clicked transition is not activated, the status line of the Renew window tells you so and nothing else is going to happen.

There are situations where a transition cannot be fired manually, although it is activated. This is the case for all transitions with an uplink. Since a transition with an uplink is waiting for a synchronization request from any other transition with the corresponding downlink, Renew cannot find such “backward” activations. You have to fire the transition with the downlink instead.

You should experiment with the simulation mode using some of the sample net systems first. Then, try to get your own reference nets to run and enjoy the simulation!

## 4.5 Simulation Server

Renew supports client/server simulations via RMI. You can set up a simulation as a Java VM of its own. You are then able to connect both locally and remotely, as long as the connection between the computers allows RMI calls (e.g. no firewall blocks them).



As a consequence of the decomposition of Renew into several plug-ins, any simulation can be published over RMI. You just need to set the appropriate properties as explained in section 2.6.8 or use the **Configure Simulation** dialog (see section 4.3.6). Therefore, this section does not focus on the configuration of a remote simulation, it just describes how to set up a simulation without using the editor's graphical user interface.

To do this, you have to export all required nets as a shadow net system first (see 4.3.1 for details). Whenever you make changes to any net of this net system, you have to generate the shadow net system again and start a new server with it.

Now you are ready to start the server itself, by issuing the following command to the Renew plug-in system:

```
startsimulation <net system> <primary net> [-i]
```

The parameters to this command have the following meaning:

**net system:** The `.sns` file, as generated in the step above.

**primary net:** The name of the net, of which a net instance shall be opened when the simulation starts. Using the regular GUI, this equals the selecting of a net before starting the simulation.

**-i:** If you set this optional flag, then the simulation is initialized only, that is, the primary net instance is opened, but the simulation is not started automatically.

As mentioned in section 2.7, the command can be passed to the plug-in system by several means. For example, to start a remotely accessible simulation with net `systemnet` out of the net system `allnets.sns` direct from the java command line, you will have to issue the following command (in Unix syntax, the `\` indicates that the printed lines should be entered as one line):

```
java -Dde.renew.remote.enable=true -jar renew5.0-Pre/loader.jar \
startsimulation allnets.sns systemnet
```

If you need a special simulation mode or any other Renew property to be configured, you can add multiple `-D` options or use one of the other configuration methods mentioned in section 2.6.1.

A simulation started by the `startsimulation` command differs slightly from a simulation started by the editor: The net loader does not look for `.rnw` files, it loads nets from `.sns` files only.

If you want to experiment with properties and commands, or if you need to pause and run the simulation interactively, you should install the Console plug-in (see section 2.7.5). When a simulation is running, several commands can be entered at the prompt to control the simulation. These commands provide the same functionality as the menu entries listed in section 4.3.6. In fact, if you use the Console plug-in in combination with the graphical editor, both command sets (menu and console) control the same simulation. The console commands are:

**simulation run:** Resumes a stopped simulation. If the `-i` option was appended to the `startsimulation` command, this command starts the simulation.

**simulation step:** Executes another simulation step. If the `-i` option was appended to the `startsimulation` command, this command executes the first simulation step.

**simulation stop:** Halts the simulation, but does not abandon it, despite of the `term` command. The `run` command continues it. This is equivalent to the menu entry `Halt simulation`.

**simulation term:** Ends and abandons the current simulation. This may result in termination of the plug-in system (see section 2.7.3).

**simulation help:** Shows a short help for all available simulation commands.



## 4.6 Error Handling

Renew helps you to maintain a syntactically correct model by making an immediate syntax check whenever an inscription has been changed. Additionally, a syntax check is done before the first simulation step of a model. The simulation will not start if there is any error in any net.

If an error is detected, an error window is opened, which displays the error message. At the bottom of the window is a button labeled **select**. Pressing this button selects the offending net element or net elements and raises the corresponding drawing. If the error originates from a text figure, that figure is edited with the corresponding text edit tool. The cursor is automatically positioned close to the point where Renew detected the error. For more information on editing see Section 4.2.2: The Text Tool.

Renew displays exactly one error at a time. If a second error is found, the old error message will be discarded and the new error message will be displayed in the error window.

Some errors are not reported at the place where they originate. E.g., if you are using a declaration figure, an undefined variable is detected where it is used, but the missing definition has to be added to declaration node. Similar effects might happen due to missing import statements. This is unavoidable, because Renew cannot tell an undeclared variable from a misspelled variable.

For some errors Renew provides a Quick Fix feature, which is described in the following section. Other errors and possible solutions are described in the subsequent sections.

### 4.6.1 Quick Fix

The Quick Fix feature improves the reporting of syntax errors by providing suitable proposals for remedies and their automatic realization.

For errors of type *No such constructor/field/method*, proposals for correct constructors, fields or methods are provided by the syntax check. If a constructor with the wrong number or types of arguments is entered, a list of existing constructor signatures is provided. If a non-existing field name for a class or an object is entered, a list of all known field names is provided. If a non-existing method is entered, a list of known method signatures where the method name is prefixed by the erroneous method name is provided. If the method name `_()` is entered, a list of all known methods is provided. For errors of type *No such variable*, type proposals are provided.

By double-clicking on one of the proposals or selecting and pressing the **apply** button, you can apply the proposed fix for the reported error. The Quick Fix changes the erroneous method/field name or constructor into the selected one or declares the variable in the declaration note. It can also automatically add import statements for unambiguous types (requires full qualified class names).

### 4.6.2 Parser Error Messages

If the expression parser detects a syntax error, it will report something like:

```
Encountered "do" at line 1, column 3.
Was expecting one of:
    "new" ...
    <IDENTIFIER> ...
```

This gives at least a hint where the syntax error originated and which context the parser expected. In our case the inscription `a:do()` was reported, because `do` is a keyword that must not be used as a channel name.

### 4.6.3 Early Error Messages

These errors are determined during the immediate syntax check following each text edit.



**Bad method call or no such method**

Typically you entered two pairs of parentheses instead of one. Possibly a class name was mistaken for a method call. Maybe a name was misspelled?

**Boolean expression expected**

An expression following the keyword `guard` must be boolean. Maybe you wrote `guard x=y`, but meant `guard x==y`?

**Cannot cast ...**

An explicit cast was requested, but this cast is forbidden by the Java typing rules. Renew determined at compile time that this cast can never succeed.

**Cannot convert ...**

The Java type system does not support a conversion that would be necessary at this point of the statement.

**Cannot make static call to instance method**

An instance method cannot be accessed statically via the class name. A concrete reference must be provided. Maybe the wrong method was called?

**Enumerable type expected**

The operator requested at the point of the error can act only on enumerable types, but not on floating point numbers.

**Expression of net instance type expected**

For a downlink expression, the expression before the colon must denote a net instance. E.g. it is an error, if in `x:ch()` the variable `x` is of type `String`. Maybe you have to use a cast?

**Expression of type void not allowed here**

An expression of void type was encountered in the middle of an expression where its result is supposed to be processed further, e.g. by an operator or as an argument to a method call. Maybe you called the wrong method?

**Integral type expected**

The operator requested at the point of the error can act only on integral types, but not on floating point numbers or booleans.

**Invalid left hand side of assignment**

In an `action` inscription, only variables, fields, and array elements can occur on the left hand side of an equation. Maybe this expression should not be an action?

**Multiple constructors match**

A constructor call was specified, but from the types of the arguments it is not clear which constructor is supposed to be called. There are overloaded constructors, but none of them seems to be better suited than the others. Maybe you should use casts to indicate the intended constructor?



**Multiple methods match**

A method call was specified, but from the types of the arguments it is not clear which method is supposed to be called. There are overloaded methods, but none of them seems to be better suited than the others. Maybe you should use casts to indicate the intended method?

**No such class**

The compiler could not find a class that matches a given class name, but it is quite sure that a class name has to occur here. Maybe you misspelled the class name? Maybe you forgot an import statement in the declaration node?

**No such class or variable**

The meaning of a name could not be determined at all. Maybe the name was misspelled? Maybe a declaration or an import statement is missing?

**No such constructor**

A matching constructor could not be found. Maybe the parameters are in the wrong order? Maybe the number of parameters is not correct? Maybe the requested constructor is not public?

**No such field**

A matching field could not be found. Maybe the name was misspelled? Maybe the requested field is not public?

**No such method**

A matching method could not be found. Maybe the name was misspelled? Maybe the parameters are in the wrong order? Maybe the number of parameters is not correct? Maybe the requested method is not public?

**No such variable**

A name that supposedly denotes a variable could not be found in the declarations. Maybe the name was misspelled? Maybe a declaration is missing?

**Not an array**

Only expressions of an array type can be postfix with an indexing argument in square brackets.

**Numeric type expected**

A boolean expression was used in a context where only numeric expressions are allowed, possibly after a unary numeric operator.

**Operator types do not match**

No appropriate version of the operator could be found that matches both the left and the right hand expression type, although both expression would be valid individually.



### **Primitive type expected**

Most operators can act only on values of primitive type, but the compiler detected an object type.

### **Type mismatch in assignment**

An equality specification could not be implemented, because the types of both sides are incompatible. One type must be subtype of the other type or the types must be identical.

### **Variable must be assignable from `de.renew.net.NetInstance`**

The variable to which a new net is assigned must be of type `NetInstance`, i.e. of exactly that type, of type `java.lang.Object`, or untyped. E.g. it is an error, if in `x:new net` the variable `x` is of type `java.lang.String`. Maybe you have to use an intermediate variable of the proper type and perform a cast later?

### **Variable name expected**

The identifier to which a new net is assigned must denote a variable. E.g. it is an error, if in `x:new net` the identifier `x` is a class name.

### **Cannot clear untyped place using typed variable**

A clear arc is inscribed with a variable that is typed. The arc is supposed to clear an untyped place. Because it cannot be safely assumed that all tokens in the place will have the correct type, it might not be possible to clear the place entirely. Consider declaring the variable that is inscribed to the arc.

### **Cannot losslessly convert ...**

A typed place must hold only values of the given type. Hence the type of an output arc expression must be a subtype of the corresponding place type. The type of an input arc expression is allowed to be a subtype or a supertype, but it is not allowed that the type is completely unrelated.

Maybe you were confused by the slight variations of the typing rules compared to Java? Have a look at Subsection 3.4.2.

### **Cannot use void expressions as arc inscriptions**

Void expressions do not compute a value. If you use such an expression, typically a method call, as an arc inscription, the simulator cannot determine which kind of token to move.

### **Class ... imported twice**

In a declaration node there were two import statements that made the same unqualified name well-known, e.g., `import java.lang.Double` and also `import some.where.else.Double`. Remove one import statement and use the fully qualified class name for that class.

### **Detected two nets with the same name**

The simulator must resolve textual references to nets by net names, hence it is not allowed for two nets to carry the same name. Maybe you have opened the same net twice? Maybe you have created new nets, which have the name `untitled` by default, and you have not saved the nets yet?



### **Flexible arcs must be inscribed**

A flexible arc is not equipped with an inscription. Flexible arcs are supposed to move a variable amount of tokens to or from a place, but this arc does not depend on any variables and lacks the required variability. Maybe you did not yet specify an inscription? Maybe the inscription is attached to the wrong net element? Maybe you want to use an ordinary arc instead?

### **For non-array inscriptions the place must be untyped**

An inscription of a flexible arc is given as a list or a vector or an enumeration, but the output place is typed. The resulting restriction on the element types could not be verified. Maybe it is possible to use an array inscription? Maybe the place should not be typed?

### **Incorrect type for flexible arc inscription**

An inscription of a flexible arc is expected to evaluate to an array or a list or a vector. It is only allowed to use enumerations on output arcs, because the elements might have to be accessed multiple times in the case of input arcs. Use an inscriptions that is correctly typed. Maybe the compiler determined the type `java.lang.Object`, but it is known that only arrays will result from the expression. In that case, use an explicit cast to indicate this fact.

### **Null not allowed for flexible arcs.**

An inscription of a flexible arc is expected to evaluate to an array or a list. The compiler was able to determine that the given expression will always evaluate to `null`. Maybe the inscription is attached to the wrong net element? Maybe the arc was not intended to be a flexible arc?

### **Only one declaration node is allowed**

You have two or more declaration nodes in your net drawing. In general, the simulator cannot determine in which order multiple declaration nodes should be processed, hence this is not allowed. Maybe a declaration node was duplicated unintentionally? Maybe you want to merge the nodes into one node?

### **Output arc expression for typed place must be typed**

A typed place must only hold values of the given type. An untyped output arc is not guaranteed to deliver an appropriate value, so this might lead to potential problems. Maybe you want to type your variables? Maybe you want to remove the typing of the place?

### **Place is typed more than once**

At most one type name can be inscribed to a place. Multiple types are not allowed, even if they are identical. Maybe a type was duplicated unintentionally?

### **Time annotations are not allowed**

The compiler detected an annotation of the form `...@...`, but the current compiler cannot handle such inscriptions, which require a special net formalism. You should switch to the `Timed Java Compiler` (see Subsection 4.3.6).

### **Transition has more than one uplink**

At most one uplink can be inscribed to a transition. Maybe an uplink was duplicated unintentionally? Maybe one uplink has to be a downlink?



### **Unknown net**

In a creation expression an unknown net name occurred. Maybe the name is misspelled? Maybe you have not opened the net in question?

### **Variable ... declared twice**

In a declaration node there were two declarations of the same variable. Remove one variable declaration.

### **Variable ... is named identically to an imported class**

In a declaration node there was a variable declaration and an import statement that referenced the same symbol, e.g., `import some.where.Name` and `String Name`. This error is rare, because by convention class names should start with an upper case letter and variable names should start with a lower case letter. You should probably rename the variable.

### **Variable of array type expected**

If a clear arc is inscribed with a typed variable, that variable should have an array type, so that the set of all tokens can be bound to the variable in the form of an array. You should check whether the correct variable is used and whether the variable is correctly typed.

## **4.6.4 Late Error Messages**

Here we discuss the error message that is not reported during the immediate check, but only during the complete check before the simulation.

### **Unsupported arc type**

An arc of the net was of an illegal type, i.e., the current net formalism does not support it. This can only happen when you execute a net with a net formalism that is incompatible with the net formalism that was used to draw the net. Maybe you should restart Renew with another net formalism?



# Bibliography

- [1] Apache Logging Services. *Log4j*, 2012.  
WWW page at <https://logging.apache.org/log4j/1.2/>.
- [2] Jonathan Billington, Sören Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The petri net markup language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003: Proceedings of 24th International Conference, Eindhoven, The Netherlands*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer-Verlag, 2003.
- [3] Lawrence Cabac. Net components: Concepts, tool, praxis. In Daniel Moldt, editor, *Petri Nets and Software Engineering, International Workshop, PNSE'09. Proceedings*, Technical Reports Université Paris 13, pages 17–33, 99, avenue Jean-Baptiste Clément, 93 430 Villetaneuse, June 2009. Université Paris 13.
- [4] Giovanni Chiola, Susanna Donatelli, and Guiliana Franceschinis. Priorities, inhibitor arcs, and concurrency in nets. In *Application and Theory of Petri Nets 1991, Proceedings 12th International Conference, Gjærn, Denmark*, pages 182–205. Springer-Verlag, 1991.
- [5] Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. Technical Report DAIMI PB-390, Aarhus University, 1992.
- [6] Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer-Verlag, 1993.  
Available at <http://www.daimi.au.dk/~sorenchr/publ.html>.
- [7] Eindhoven University of Technology. *Woflan – The Workflow Analyser*, 1998.  
WWW page at <http://www.win.tue.nl/woflan/>.
- [8] Erich Gamma. *JHotDraw*, 1998.  
Available at <http://members.pingnet.ch/gamma/JHD-5.1.zip>.
- [9] Olaf Kummer. Simulating synchronous channels and net instances. In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, editors, *5. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 73–78. Forschungsbericht 694, Universität Dortmund, Fachbereich Informatik, October 1998.
- [10] Olaf Kummer. Tight integration of Java and Petri nets. In J. Desel and A. Oberweis, editors, *6. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 30–35. J.W. Goethe-Universität, Institut für Wirtschaftsinformatik, Frankfurt am Main, Fachbereich Informatik, October 1999.



- [11] Olaf Kummer. *Referenznetze*. Logos-Verlag, Berlin, 2002.
- [12] Charles A. Lakos and Søren Christensen. A general systematic approach to arc extensions for coloured petri nets. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 1994.  
Available at <http://www.daimi.au.dk/~sorenchr/publ.html>.
- [13] Peter Langner, Christoph Schneider, and Joachim Wehler. Petri net based certification of event-driven process chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri Nets 1998: Proceedings of 19th International Conference, Lisbon, Portugal*, volume 1420 of *Lecture Notes in Computer Science*, pages 286–305. Springer-Verlag, 1998.
- [14] Carl Adam Petri. Introduction to general net theory. In Brauer, W., editor, *Net Theory and Applications, Proc. of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979*, volume 84 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 1980.
- [15] QoS.ch Sarl (Switzerland). *Reload4j*, 2022.  
WWW page at <https://reload4j.qos.ch/>.
- [16] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1–2):1–34, 1991.
- [17] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer-Verlag, 1998.
- [18] Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 1998.
- [19] Peter van der Linden. *Just Java*. The Sunsoft Press Java Series. Prentice Hall, 1996.



## Appendix A

# Contacting the Team

To get in contact with us, you can send an email to

`support@renew.de`

regarding any aspect of the Renew tool, especially update notification requests, bug reports, feature requests, and source code submissions. Our postal address is

Arbeitsbereich ART

— Renew —

Fachbereich Informatik, Universität Hamburg

Vogt-Kölln-Straße 30

D-22527 Hamburg

Germany

in case you do not have access to email. The latest news about Renew are available from the URL

`http://www.renew.de/`

and in the same place improved versions and bug fixes appear first.



# Appendix B

## File Types

Renew creates, uses and consists of many different kinds of files that are distinguished by their file extension. We will describe the most important file types here.

A file named `net` contains

<code>*.aut</code>	an autosaved net drawing (may be renamed to <code>*.rnw</code> )
<code>*.bak</code>	a backup net drawing (may be renamed to <code>*.rnw</code> )
<code>*.bat</code>	a Windows batch file
<code>*.bib</code>	a BibTeX bibliography
<code>*.class</code>	a compiled Java class
<code>*.draw</code>	a drawing (mime-type: <code>application/x-renew-drawing</code> )
<code>*.dtd</code>	an XML DTD
<code>*.dvi</code>	a device independent output file of TeX
<code>*.eps</code>	Encapsulated PostScript graphics
<code>*.gif</code>	a bitmap in GIF format
<code>*.jar</code>	a JAR compressed archive
<code>*.java</code>	Java source code
<code>*.jj</code>	a JavaCC grammar
<code>*.pdf</code>	a PDF document
<code>*.pnml</code>	one or more nets in PNML format
<code>*.rnw</code>	a net drawing (mime-type: <code>application/x-renew-net</code> )
<code>*.rst</code>	a saved simulation state (mime-type: <code>application/x-renew-state</code> )
<code>*.sns</code>	a serialized shadow net system (mime-type: <code>application/x-renew-sns</code> )
<code>*.stub</code>	a net stub description
<code>*.sty</code>	a TeX style definition
<code>*.tex</code>	a TeX source file
<code>*.zip</code>	a ZIP compressed archive



## Appendix C

# Keyboard Shortcuts

Shortcuts listed here with the **Ctrl** modifier key should be used on computers running MacOS with the **Cmd** modifier key instead .

### List by Category

(**a**) Apple Specific, (**b**) build-in, (**c**) customizable, (**r**) reserved, (**s**) switching (simulation/editing), (**p**) provided by (optional) plugin

Modifier	Key	Function	%	Plugin
<b>File</b>				
Ctrl	N	New RNW	b	
Ctrl-Shift	N	Open Navigator	p	Navigator
Ctrl	O	Open Drawing	b	
Ctrl	S	Save Drawing	b	
Ctrl-Shift	S	Save Drawing As...	b	
Ctrl	W	Close Window	b	
Ctrl-Shift	W	Close All Windows	b	
Ctrl	E	EPS Export	p	Export
Ctrl-Shift	P	PDF Export	p	Export
Ctrl	9	PNG Export	p	Export
Cmd	Q	Quit Program	ra	
<b>Edit</b>				
Ctrl	Z	Undo	b	
Ctrl	Y	Redo	b	
Ctrl	X	Cut	b	
Ctrl	C	Copy	b	
Ctrl	V	Paste	b	
Ctrl	D	Duplicate	b	
Ctrl	F	Search	b	
Ctrl	G	Search & Replace	b	
Ctrl	A	Select All	b	
Ctrl	1	group Net Component	p	Net Components
Ctrl	3	ungroup Net Component	p	Net Components
Ctrl	Enter	Finish Editing	b	
Ctrl	B	Send to Back	b	
Ctrl-Shift	B	Bring to Front	b	
Ctrl	\	Align Center	c	
Ctrl-Shift	-	Align Middles	c	
Ctrl	=	Spread Centers	c	



Ctrl	;	Spread Middles	c	
Ctrl	/	Spread Diagonal	b	
<b>Net</b>				
Ctrl-Option	B	Set Breakpoint	bs	
Ctrl-Option	C	Clear Breakpoint	bs	
Ctrl-Option	S	Syntax Check	b	
Ctrl-Option	L	Layout Check	b	
<b>Simulation</b>				
Ctrl	R	Run Simulation	b	
Ctrl	I	Simulation Step	b	
Ctrl-Shift	I	Simulation Net Step	b	
Ctrl	H	Halt Simulation	b	
Cmd-Shift	H	Halt Simulation	ba	
Ctrl	T	Terminate Simulation	b	
Ctrl	L	Show Simulation Trace	p	Logging
Ctrl-Option	B	Set Breakpoint	bs	
Ctrl-Option	C	Clear Breakpoint	bs	
<b>Without Menu Entry</b>				
Ctrl	+ or mousewheel up	Zoom in	b	
Ctrl	- or mousewheel down	Zoom out	b	
Ctrl	0	Reset Zoom Factor	b	

## Alphabetical List

(a) Apple Specific, (b) build-in, (c) customizable, (r) reserved, (s) switching (simulation/editing), (p) provided by (optional) plugin

Modifier	Key	Function	%	Plugin
Ctrl	+	Zoom in	b	
Ctrl	-	Zoom out	b	
Ctrl-Shift	-	Align Middles	c	
Ctrl	/	Spread Diagonal	b	
Ctrl	0	Reset Zoom Factor	b	
Ctrl	1	group Net Component	p	Net Components
Ctrl	3	ungroup Net Component	p	Net Components
Ctrl	9	PNG Export	p	Export
Ctrl	;	Spread Middles	c	
Ctrl	=	Spread Centers	c	
Ctrl	A	Select All	b	
Ctrl	B	Send to Back	b	
Ctrl-Shift	B	Bring to Front	b	
Ctrl-Option	B	Set Breakpoint	bs	
Ctrl-Option	B	Set Breakpoint	bs	
Ctrl-Option	C	Clear Breakpoint	bs	
Ctrl-Option	C	Clear Breakpoint	bs	
Ctrl	C	Copy	b	
Ctrl	D	Duplicate	b	
Ctrl	E	EPS Export	p	Export
Ctrl	F	Search	b	
Ctrl	G	Search & Replace	b	
Ctrl	H	Halt Simulation	b	



Cmd-Shift	H	Halt Simulation	ba	
Ctrl	I	Simulation Step	b	
Ctrl-Shift	I	Simulation Net Step	b	
Ctrl	L	Show Simulation Trace	p	Logging
Ctrl-Option	L	Layout Check	b	
Ctrl	N	New RNW	b	
Ctrl-Shift	N	Open Navigator	p	Navigator
Ctrl	O	Open Drawing	b	
Ctrl-Shift	P	PDF Export	p	Export
Cmd	Q	Quit Program	ra	
Ctrl	R	Run Simulation	b	
Ctrl	S	Save Drawing	b	
Ctrl-Shift	S	Save Drawing As...	b	
Ctrl-Option	S	Syntax Check	b	
Ctrl	T	Terminate Simulation	b	
Ctrl	V	Paste	b	
Ctrl	W	Close Window	b	
Ctrl-Shift	W	Close All Windows	b	
Ctrl	X	Cut	b	
Ctrl	Y	Redo	b	
Ctrl	Z	Undo	b	
Ctrl	\	Align Center	c	
Ctrl	Enter	Finish Editing	b	

## Toggle Key Mappings for Main Drawing Tools

By setting the property `de.renew.keymap.use-mapping=true` keystrokes can be used to select drawing tools (tool buttons). This mechanism can easily be customized by setting options in the form of `de.renew.keymap.XY` where `XY` is the name of the tool; for example `de.renew.keymap.Transition_Tool=t`. Set these options in your configuration files in order to be activated during GUI initialization. Use the *Shift* modifier key to select a tool permanently. The following keys constitute a preset selection for the main Renew drawing tools:

.	Selection Tool
t	Transition Tool
p	Place Tool
v	Virtual Place Tool
a	Arc Tool
x	Test Arc Tool
r	Reserve Arc Tool
f	Flexible Arc Tool
i	Inscription Tool
n	Name Tool
d	Declaration Tool



# Appendix D

## License

‘We’ refers to the copyright holders. ‘You’ refers to the licensee. ‘Renew’ refers to the complete set of sources, executables, and sample nets that make up the Reference Net Workshop.

Renew is available free of charge, but not without restrictions. The majority of Renew is published under the GNU Lesser General Public License (for details see Section “Original Parts” below). However, Renew builds up on other people’s work that has been licensed under other terms.

The license section got a bit long. We apologize, but we cannot hope to do better, because we included many external parts with many different licenses.

### D.1 Contributed Parts

Renew uses several parts that were previously developed by other people and have been made publicly available.

#### D.1.1 The collections Package

The `collections` package is used as our `set/queue/list` implementation. The relevant license information states:

Originally written by Doug Lea and released into the public domain.

You can use it as you want. Please note that Doug Lea now suggests to use the container libraries that come with Java 1.2 instead of his own libraries. We are currently working on the migration, so far all plug-ins except the FS plug-in have been adopted to the Java container classes.

#### D.1.2 The JHotDraw Package

The `JHotDraw` graphical editor written by Erich Gamma is copyrighted. The relevant license information states:

JHotDraw is copyright 1996, 1997 by IFA Informatik and Erich Gamma.

It is hereby granted that this software can be used, copied, modified, and distributed without fee provided that this copyright notice appears in all copies.

#### D.1.3 Code Generated from JavaCC

Some of the code of Renew was generated by the parser generator JavaCC. JavaCC is distributed under the 3-Clause BSD license, a copy of which can be found here <https://opensource.org/licenses/BSD-3-Clause>. The relevant license information states:



Copyright (c) 2006, Sun Microsystems, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Sun Microsystems, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Hence we would like to explicitly point out that Sun is not responsible for any problems that might result from the use of the output of JavaCC.

#### D.1.4 Bill's Java Grammar

A Java grammar `billsJava1.0.2.jj` was distributed together with JavaCC 0.7 as a sample grammar. Bill McKeeman ([mckeeman@mathworks.com](mailto:mckeeman@mathworks.com)) contributed this grammar to JavaCC. The relevant license information from Sun states:

##### 2. SAMPLE GRAMMARS

You may modify the sample grammars included in the Software to develop derivatives thereof ("Sample Grammar Derivatives"), and sublicense the Sample Grammar Derivatives directly or indirectly to your customers.

##### 4. YOUR INDEMNIFICATION OF SAMPLE GRAMMARS DERIVATIVES AND DEVELOPED PRODUCTS

You agree to indemnify, hold harmless, and defend Sun from and against any claims or suits, including attorneys' fees, which arise or result from any use or distribution of Sample Grammar Derivatives and/or Developed Programs.

The original parts of `billsJava1.0.2.jj` which are now contained in a modified form in the files `JavaNetParser.jj`, `FSNetParser.jj`, `FSParser.jj`, and `StubParser.jj` are Copyright (C) 1996, 1997 Sun Microsystems Inc. A sublicense for these grammars is hereby granted. If you have any further questions, please consult the file `COPYRIGHT` as distributed with JavaCC.

#### D.1.5 Graph Layout Algorithm

The graph layout algorithm used in the class `de.renew.gui.GraphLayout` was originally provided by Sun as part of the Java Development Kit. The relevant license information from Sun states:

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

...

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.



We would like to explicitly point out that Sun is not responsible for any problems that might result from the use of the graph layout algorithm. See the source files for Sun's original disclaimer.

### D.1.6 The Reload4j Package

Renew comes with an unmodified, binary distribution of **Reload4j**, which is a fork of Apache **log4j** version 1.2.17. **Reload4j** is licensed under the Apache License, Version 2.0. The full license is included in the distribution, a copy of the License may also be obtained from <https://www.apache.org/licenses/LICENSE-2.0>. The relevant license information states:

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - ...You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.  
...
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.



8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

### D.1.7 The FreeHEP VectorGraphics package

Renew comes with a slightly modified version of Java libraries provided by the the FreeHEP project to support several export formats like EPS, PDF, SVG or PNG. The FreeHEP libraries are provided under the same license as Renew itself (the GNU Lesser General Public License, for a link see below) which allows the inclusion of a binary distribution within other projects. The modified sources are available at <https://github.com/renew-tgi/>.

### D.1.8 JLine2

Renew comes with an unmodified, binary distribution and some slightly modified classes of the JLine2 package, which provides the basis for the Console plug-in. JLine2 is distributed under the BSD License, meaning that you are completely free to redistribute, modify, or sell it with almost no restrictions. The full license is included in the distribution, a copy of the License may also be obtained from <https://opensource.org/licenses/bsd-license.php>.

The original JLine2 package can be obtained from <https://github.com/jline/jline2/>, the modified classes are part of the source distribution of Renew.

### D.1.9 Commons CLI

Commons CLI is a library that supports parsing command line commands in Java and is distributed under the Apache License, Version 2.0 as of 2004. It is used by Renew to provide a uniform command line interface. The Commons CLI library is available at the Apache project website <https://commons.apache.org/cli>.

### D.1.10 DockingFrames

Renew comes with an unmodified, binary distribution of the DockingFrames framework which is used for the graphical user interface of Renew. The framework is provided under the same license as Renew itself (the GNU Lesser General Public License, for a link see below).

### D.1.11 Other Libraries

Other libraries might provide their own licenses.

## D.2 Original Parts

This copyright section deals with those parts of Renew that are not based on other works, i.e. the example nets and the packages `fs` and `de.renew` without the JavaCC grammars.



### D.2.1 Example Nets

The example nets are in the public domain. You may modify them as you like. You may use them as the basis for your own nets without restrictions.

### D.2.2 Java Source Code and Executables

Sources and executables are copyright 1998–2026 by Olaf Kummer, Frank Wienberg, Michael Duvignau, Lawrence Cabac, Michael Haustermann, David Mosteller and others. You can distribute these files under the GNU Lesser General Public License.

You should have received a copy of the GNU Lesser General Public License along with this program in the file `doc/COPYING`; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

## D.3 Created Parts

You are permitted to use works that you create with Renew (i.e., Java stubs, net drawings, EPS/PDF output, simulation states, and other exported data) without restrictions.

## D.4 Disclaimer

We distribute Renew in the hope that it will be useful, but *without any warranty*; without even the implied warranty of merchantability or fitness for a particular purpose.

We are *not liable* for any direct, indirect, incidental or consequential damage including, but not limited to, loss of data, loss of profits, or system failure, which arises out of use or inability to use Renew or works created with Renew. This clause does not apply to gross negligence or premeditation.

Some parts of Renew may use patented techniques that may not be freely usable in some countries. In that case, it is the responsibility of the user of Renew to obtain a license on the aforementioned techniques before using Renew.

Some parts of Renew may include additional disclaimers in their license terms. In such cases, both disclaimers hold simultaneously. If one clause of any disclaimer is found invalid under applicable law, this does not affect the validity of the remaining clauses or of other disclaimers.

The applicable court is Hamburg, Germany.

## D.5 Open Source

This license is intended to be Open Source compliant.

If you find any clause within this license that is incompatible with the guidelines set forth in the Open Source definition (see <https://opensource.org/osd>), please contact the authors.