Renew – User Guide

Olaf Kummer

Frank Wienberg

Michael Duvigneau

University of Hamburg
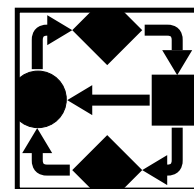
Department for Informatics

Theoretical Foundations Group

Distributed Systems Group

This document was prepared using the LaTeX typesetting system.

Contents

3

List of Figures

List of Tables

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 that will serve as a more in-depth introduction to Java. [15] is a good first start for experienced programmers.

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 modelling 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 1.3 If you are already familiar with a previous version of Renew, you should simply skim the manual and look for the Renew 1.3 icons as shown to the left. The paragraphs that are tagged with this icon elaborate on new features of the current version.
You should also consult Section 2.3 for some notes on the upgrade process. The upgrade might require some explicit actions on your part.

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 like to thank Berndt Farwer for the preparation of the Macintosh version. We are indepted to the authors of various freeware libraries, namely Erich Gamma, Doug Lea, David Megginson, Bill McKeeman and Sriram Sankar.

Valuable contributions and suggestions were made by students at the University of Hamburg, most notably Lars Braubach, Matthias Ernst, Olaf Großler, Sven Heitsch, Michael

Köhler, Annette Laue, Matthias Liedtke, Marcel Martens, Klaus Mitreiter, Martin Pfeiffer, Alexander Pokahr, Heiko Rölke, and Eberhard Wolff.

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

Before you proceed, make sure to have a system backup available on the off-chance that an error occurs during the installation procedure.

You must have Java 1.1 or Java 1.2 installed. If you have not done this yet, we suggest that you get the latest Java Runtime Environment from Sun at the URL

`http://www.javasoft.com/products/index.html`

where versions for Windows and Solaris are available. If you use OS/2, have a look at IBM's Software Choice pages at

`http://service.boulder.ibm.com/asd-bin/doc/`

and for Apple Macintosh there is a product called Mac OS Runtime for Java available from

`http://www.apple.com/java/`

currently in version 2.0. For Linux,

`http://www.blackdown.org/`

will help you. All runtime environments are available free of charge. We recommend the latest version of JDK 1.1, which at this point of time should be JDK 1.1.7 or 1.1.8, depending on the operating system.

If you intend to use the XML file format as described in subsection 4.4.1, you should download an XML parser. By default, Renew is configured to XML4J by Alphaworks/IBM. At

`http://www.alphaworks.ibm.com/tech/xml4j`

you can download it for free. You must add XML4J to your CLASSPATH before the XML import works correctly.

2.2　Possible Collisions

While Renew is based on the `collections` package by Doug Lea [10] and the `JHotDraw` package by Gamma [5], both packages are distributed with Renew. In the case of the `collections` package, this is done for convenience, but the `JHotDraw` 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 2.8 you can find a list of differences, if you are interested in further details.

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 1.1

Serialized shadow net systems exported by Renew 1.1 cannot be used with any later versions. You can simply reexport the net system.

Stubs compiled with Renew 1.1 cannot be used with any later version. You can simply recompile the stubs and the resulting Java files.

2.4　Installing Renew

The two jar-files `renew1.3base.jar` and `renew1.3source.jar` 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. For platforms where `jar` is not available or not convenient, we also provide `zip`-files.

The base Renew distribution consumes about 5.5 MByte. The source distribution adds another 6 MByte to that figure.

If you use an Apple Macintosh, refer to subsection 2.5.2, where we provide special installation notes.

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 in DOS-based environments, the : is reserved for drive letters, so ; is used for lists.

To extract the base distribution, issue the command

```
jar xf renew1.3base.jar
```

or

```
unzip renew1.3base.zip
```

12

if your Java environment does not support the `jar` command.

A directory `renew1.3` will be created in the current directory. Documentation files, e.g. this manual and the GNU GPL, are placed in the subdirectory `renew1.3/doc`. The subdirectory `renew1.3/samples` contains example nets. The file `renew1.3/renew.jar` is an uncompressed `jar`-file that should be added to your class path, e.g., by saying

```
setenv CLASSPATH .:/some/where/renew1.3/renew.jar
```

if you extracted the `jar`-file into `/some/where`. In a DOS-based environment, this would look something like

```
set CLASSPATH=.;C:\some\where\renew1.3\renew.jar
```

(mind the drive letter and the use of backslash instead of slash and semicolon instead of colon).

Once you have set up the classpath correctly, you only have to type

```
java de.renew.gui.CPNApplication
```

to start Renew. Note that for Unix, OS/2, and Windows 95/98/NT we provide ready-made startup scripts already. In that case, you do not have to set the classpath manually. 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.

To extract the source distribution, issue the command

```
jar xf renew1.3source.jar
```

which will put files into the directories `renew1.3/CH`, `renew1.3/collections`, `renew1.3/fs`, and `renew1.3/de`. In this case you should point your `CLASSPATH` to this source directory (and not to the `renew.jar`-file!) and make a verifying compilation of the Java sources. If you succeed, you can delete the file `renew1.3/renew.jar` with the original `class`-files.

If you are using Java 1.2, you must uncomment some lines in the source files

```
CH/ifa/draw/util/NullGraphics.java
CH/ifa/draw/util/ContextGraphics.java
de/renew/util/PostscriptWriter.java
```

These comments hide the method `drawString(AttributedCharacterIterator,int,int)` from Java's view, because this method is not allowed under Java 1.1. The method is always declared at the very end of the respective files. After these changes, only some deprecation warnings remain, which do not stop Renew from running correctly.

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 start some other modes as summarized in Table 2.1.

In the next subsections we will only describe the usage of the basic script, since the other scripts behave very similarly.

13

script name	invoked mode
renew	concurrent simulation mode
srenew	extended sequential simulation mode
	see subsection 2.6.6 for details

Table 2.1: The startup scripts

2.5.1 OS/2

On an OS/2 Warp system, uncompress the archive `renew1.3base.jar` as described before. You can now change to the directory `renew1.3\bin\os2`. There you can find the Rexx script `InstallRenew.cmd` that will automatically create a startup script and a desktop folder for Renew. Run the script either by double-clicking the `InstallRenew.cmd` icon or by invoking `InstallRenew` on an OS/2 command line in the directory mentioned above. Now, a `Renew 1.3` folder should appear on your desktop, containing program objects for Renew in the various modes. Also, shadows of the documentation and the samples folder reside here. If the folder does not appear, run the install command from the command line and look for error messages. If all else fails, try to install Renew manually as described in the previous section.

The Renew program object is associated to all files with the `RNW` extension, so that you can start Renew with a certain drawing by opening the corresponding file in the WorkPlaceShell. You should not try to open many files at the same time in this manner, though, as the WPS starts the associated program for each file, not only once with all the files.

Renew 1.1 If you want to load drawings into a running Renew by double-clicking them in the WPS, we provide a new feature for that.

Renew can set up a so-called Drawing Load Server (see Subsection 2.6). For OS/2, there is a Rexx program called `loadrenew.cmd` that acts as a client and tells the server to load a drawing. To automatically set up Renew as a Drawing Load Server and install a client, use `InstallRenew.cmd` from the command line and give an unused TCP/IP port number as the sole parameter, like this:

```
InstallRenew 1234
```

Then, an additional program object referring to `loadrenew.cmd` is created in the `Renew 1.3` folder and is also associated to the `RNW` file extension. Because this program object is created first, the default operation (double-click) for `RNW` files is now to start the Drawing Load Client, which only works if Renew is already running. To open a drawing while Renew is not running, you can still invoke the drawing's context menu (right-click) and select `Renew 1.3`. Again, this works only for single drawings.

2.5.2 Apple Macintosh

For Macs, we suggest that you do not start with the standard archive, but use the archive `Renew1.3.sea.hqx` instead. You will find two Renew applications in the directory `Renew1.3` after decompression. Start the `Renew` application for the standard simulation mode.

The documentation and the example files are included, too.

The current Mac OS has a Java runtime installed already. If you use an older version, make sure to install the Mac OS Runtime for Java before you install Renew, because it is not included in our archive.

2.5.3 Unix

We supply a simple install script at `renew1.3/bin/unix/installrenew` that will handle the installation on most flavors of Unix. Run that script with

14

```
cd renew1.3/bin/unix
sh installrenew
```

and it will create the shell scripts `renew`, `compilestub`, `makestub`, and some scripts for the additional Renew mode in the same directory. Calling the script `renew` will start the net editor. See Subsection 3.8.1 for further explanations of the commands `compilestub` and `makestub`.

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` directory, otherwise it cannot find the location of the package.

2.5.4 Windows

For Windows we provide simple startup batch files for the basic simulation mode (`RENEW.BAT`) and other modes in the directory `renew1.3\bin\win`.

```
cd renew1.3\bin\win
renew
```

The batch files must be started from their own directory.

We also provide an installation script at `renew1.3\bin\win\installrenew` for your convenience. This batch file has to be started from its own directory, too.

```
cd renew1.3\bin\win
installrenew
```

This will adapt the startup batch files, so that they can be started from every directory.

If this does not work, edit the the classpath according to your installation.

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 Drawing Load Server

Renew 1.1 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 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.

By providing a simple command line parameter, you can tell Renew to listen to requests to load a certain drawing on a TCP/IP socket. Just specify an unused TCP/IP port number, say 1234, by setting

```
java -Dde.renew.loadServerPort=1234 ...
```

When Renew is started with this command line parameter, 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). Now, all you need is such a simple client application that opens a socket to the server (using the same port number as above), transmits its parameter, and closes the socket again. We provide such a client application as a Rexx program for OS/2 (see above). If you are brave and not an OS/2 user, try to write a drawing load client for other platforms (e. g. in `C`) and make it available for the next Renew release! Perhaps we will do this ourselves later on, but there is also the Java Drag-and-Drop-API...

If you have a drawing load client for your operating system, we suggest to associate the client to the `RNW` extension and start Renew once manually. When a drawing is double-clicked in the file manager, the client is invoked and can transmit the drawings 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.

2.6.2 Redraw Strategy

By default, Renew uses a double buffering algorithm to perform display updates, in order to reduce flickering of the screen.

In some cases, this can expose a Java 1.2 bug that leads to bad screen output. Therefore, the double buffering should be disabled for Java 1.2 and later versions of Java. Usually, this is done automatically but you can force this behavior using

```
java -DCH.ifa.draw.specialUpdate=true ...
```

if you experience redraw problems under Java 1.1. If this is the case, please let us know.

If you want to try buffered display updates using Java 1.2, issue the command

```
java -DCH.ifa.draw.specialUpdate=false ...
```

and in many cases the flickering will reduce without any negative effects.

2.6.3 Resizable Main Window

If your Renew window looks quite different than in Figure 4.1, you probably encountered a Java bug in the `pack()` method of `java.awt.Frame` present in (most?) MS Windows Java versions. Since the Renew window normally is not resizable, this bug can be quite annoying. But we provide a workaround: On MS Windows systems, the Renew window should automatically be resizable, so that you can adjust its size yourself. If the bug is not detected automatically, you can start Renew with the additional command line parameter

```
java -Dde.renew.windowResizable=true ...
```

If the bug is detected mistakenly, i.e., the window is displayed correctly, but is still resizable, use

```
java -Dde.renew.windowResizable=false ...
```

Some X-window managers ignore the setting of a window to not be resizable, so we can't do anything about this bug there.

On some X-Window managers it has been observed that the main window jumps off the visible screen area after its creation, so that its cannot be accessed. This behavior, too, disappears when you use the option

```
java -Dde.renew.windowResizable=true ...
```

to make the main window resizable. Regarding the explanation of this Java bug, we have no clue.

2.6.4 Multiprocessor Mode

Renew
1.1   We have added support for shared-memory multiprocessors to Renew. Depending on your specific application, 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.

In order to run Renew on a multiprocessor hardware you need a Java implementation that supports native threads. Note that most Java implementations support only so-called green threads, which cannot exploit hardware parallelism. You must then add the command line parameter

```
java -Dde.renew.simulatorMode=2 ...
```

if your want to start two concurrent simulation threads. Replace the number 2 by a greater number, if you want to start more simulation threads. Note that it will usually be detrimental to the performance, if you increase the number beyond the number of physical processors.

Before using multiple processors, you should probably try to optimize performance with other means. You should use a Java implementation with a just-in-time compiler. You should disable the trace mode for all or most net elements, so that unneeded trace messages are eliminated. You should open only very few net instance windows, so that the graphical representation of the markings does not need to be updated.

2.6.5   Sequential Mode

By adding the command line parameter

```
java -Dde.renew.simulatorMode=-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 3.8 cannot be used in this mode.

2.6.6   Extended Sequential Mode

Renew 1.2 If we restrict our 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 3.9.2 and 3.9.3.

The extended sequential mode can be set up by providing the argument

```
java -Dde.renew.mode=de.renew.gui.SequentialJavaMode ...
```

at the command line. This mode allows you to draw and simulate the extended arc types and it automatically sets the simulator mode. If both parameters are set, `de.renew.mode` overrides `de.renew.simulatorMode`. You can also reach this mode by using the command `srenew` (sequential renew) if you are using the predefined shell scripts.

Please note that net methods as described in Section 3.8 cannot be used in this mode.

2.6.7   Class Reloading

Renew 1.2 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 custom class loader to Renew.

By adding the command line parameter

```
java -Dde.renew.classReinit=true ...
```

you can request that all user classes that are referenced by a net are reloaded before every compilation of a net. Note that this mechanism may result in some problems when you access the Java reflection API.

Because the reloading of classes might affect performance, it is disabled by default.

2.7   Troubleshooting

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

- I cannot extract the files from the archives.

  Maybe the files got corrupted on their way to you? Maybe you are using an old version of `unzip`? If you have a version of the JDK that does not support zipped `jar` archives, please let us know.

  Note that you must use Java 1.1 or newer to use `jar` archives and that Renew requires Java 1.1 for different reasons anyway.

- Java is not found.

  Probably the shell scripts try to look for Java in the wrong places.

- Java cannot find the class `de.renew.gui.CPNApplication`.

  Have you set the environment variable `CLASSPATH` correctly? Maybe it is set incorrectly in your `.login`, `AUTOEXEC.BAT` or equivalent shell configuration files? You can find out the current value of the `CLASSPATH` variable by issuing `echo $CLASSPATH` (Unix) or `echo %CLASSPATH%` (DOS). Be sure to issue this command in the same environment in which you tried to start Renew!

  If your `CLASSPATH` seems to be correct, try extracting the class files from `renew.jar` and add the main directory to your class path. Maybe your Java implementation cannot access `jar`-files.

- Renew starts, but the window titles are incorrect under the X Windows System.

  Try a different window manager, e.g., `mwm` is known to work correctly. This is a general Java problem and not related to Renew, so we cannot do anything about it.

- My Renew window is too large or too small, so that is consumes too much space or (even worse) the tools are not reachable.

  See subsection 2.6.3 for a workaround.

- The Renew window does not appear or it does appear, but jumps off the screen after a short time.

  See subsection 2.6.3 for a workaround. Alternatively, under Unix you might try a different window manager.

- Renew does not properly display the contents of a drawing. Renew redraws only part of the screen.

  See subsection 2.6.2 for a workaround.

- I cannot open the sample files.

  Sometimes you need to add the root directory / (or \, depending on your operating system) to your class path.

- I cannot compile Renew under Java 1.2.

  Due to some changes in the Java API it is difficult at this point of time to create a version of Renew that compiles under Java 1.1 as well as under Java 1.2. Since Java 1.2 is not yet available for all platforms, we decided to keep Renew compatible with Java 1.1 for the time being.

  You can make the source compatible with Java 1.2 yourself by following the instrcutions from Subsection 2.4.2.

2.8 History

Version 1.0 was the first public release. It included a net editor, a reference net simulator, a Java stub compiler, and example nets.

2.8.1 Changes in Version 1.1

Modifications

Some performance enhancements were implemented and minor bugs were fixed. Some source level inconsistencies were cleaned up. The thread model of Java 1.2 was adopted. The source code was changed to be compilable with Java 1.1.3. The windowing code was made more robust under Java 1.2.

The handling of `null`-objects in the simulator was corrected. The type system was made more compatible with the Java type system. The trace flag of net elements is now saved to disk. The simulation performance was improved. The garbage collection of net instances was improved.

The graphical user interface was improved for some window managers. The presentation of current markings was improved. The interactive execution of reference nets has been improved a lot (see Section 4.4.5).

Additions

The parallel simulation code was added. The checks for double names and for cyclic channel dependencies were added. Transition inscriptions may now include several parts separated by semicolons. Virtual places may now be used in nets.

During the simulation, bindings can be selected and fired under user control. The multiset of tokens contained in a place instance can be displayed as just the cardinality of the multiset, a collection of all tokens in the multiset directly within the drawing, or in a separate window. Individual components of tuples can be inspected. Initial markings are hidden during the simulation.

2.8.2 Changes in Version 1.2

Modifications

The simulation engine was made more robust and flexible. Minor bugs were fixed.

A single inscription figure may now contain multiple arc inscriptions or initial marking inscriptions that are separated by semicolons. Slight inconsistencies in the inscription language were cleaned up. The type rules were improved. The results of action inscriptions may now be passed through synchronous channels even in the presence of typed variables.

Some display problems with Java 1.2 have been fixed.

Additions

Flexible arcs were added. Clear arcs were added. Inhibitor arcs were added.

Marked places and firing transitions can now be highlighted during the simulation. A rudimentary net layout algorithm has been implemented. The state of a running simulation can now be saved and restored. Restarting a simulation may now reload Java classes.

Export of Encapsulated PostScript was implemented. Selection of groups of figures was improved.

2.8.3 Changes in Version 1.3

Modifications

Some minor improvements of the graphical user interface were applied.

Additions

The timed simulation mode was added. Lists were provided in addition to tuples.

Breakpoints were added in order to control the graphical simulation. An XML import and export facility was added. A graph layout algorithm that may help in viewing nets was added. More commands for arranging figures manually were provided. The ability to select and deselect figures by type was added.

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 [6] and [7]. Finally, we are going to see how nets and Java can seamlessly interact with each other.

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 [3].

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 3.9.

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.
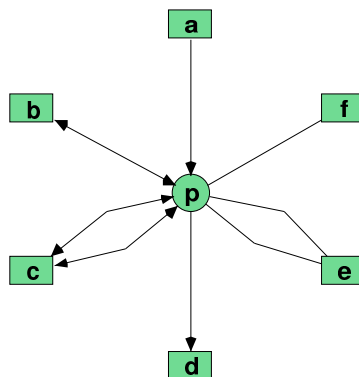
Figure 3.1: The net `elements`

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`.
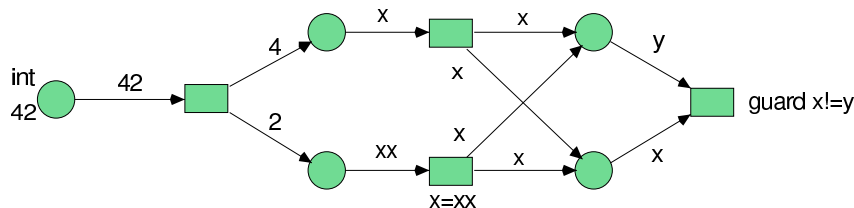
Figure 3.2: The net `colored`

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 $x \neq y$, 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.

`boolean`	boolean values (`true`, `false`)
`byte`	8-bit signed integers
`short`	16-bit signed integers
`int`	32-bit signed integers
`long`	64-bit signed integers
`char`	16-bit unsigned Unicode characters
`float`	32-bit IEEE floating point numbers
`double`	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 looses 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 modelling 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 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 it 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

23
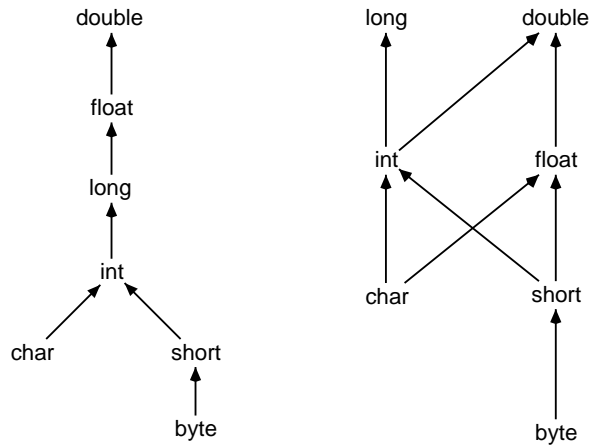
Figure 3.3: The Java type hierarchy and the hierarchy of lossless conversions

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 `!=`. 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 afterwards, 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 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

*	multiply	number
/	divide	number
%	modulo	number
+	plus	number, `String`
–	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
ˆ	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

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.
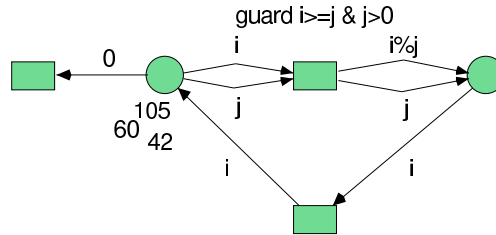
Figure 3.4: The net gcd

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.
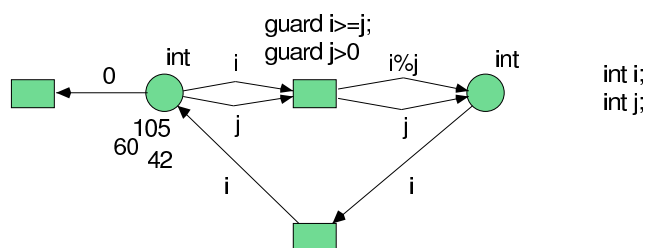
Figure 3.5: The net `gcdtyped`

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.

Renew
1.1 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.
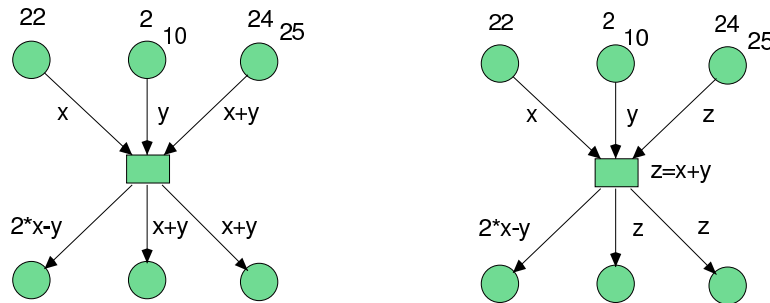
Figure 3.6: The net `equality`

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 backwards. 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 backwards. If due to an equality specification the result of such a cast is known, it is propagated backwards to the casted expression, possibly after some conversion.

If a backwards computation is desired in the other cases, it has to be made explicit. In our example, we could complement the equation x=y+z by y=x-z and z=x-y. 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.
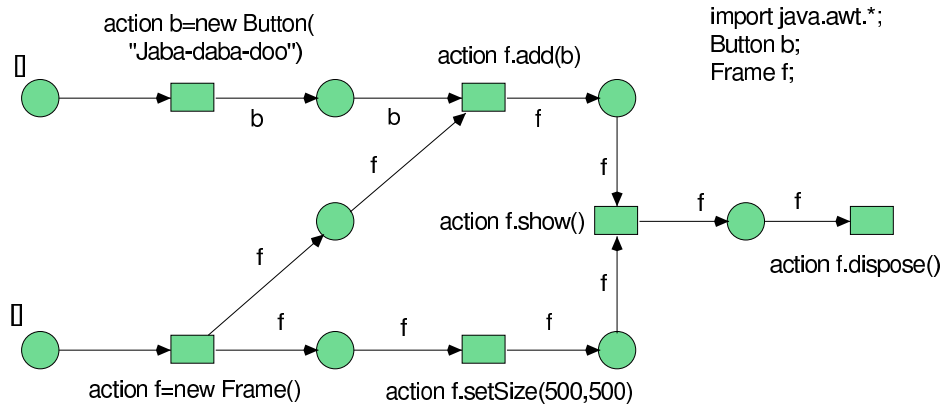
Figure 3.7: The net `frame`

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"`,

29

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).
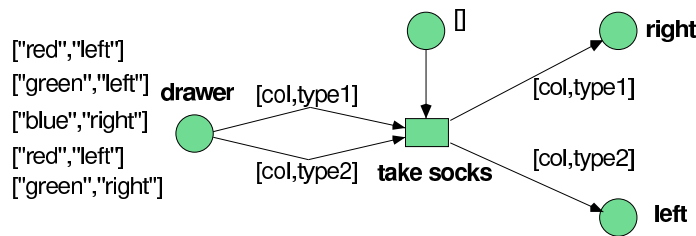
Figure 3.8: The net `socks`

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.

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.
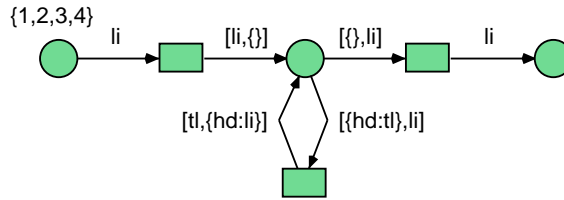
Figure 3.9: The net `reverse`

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 orginal 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.
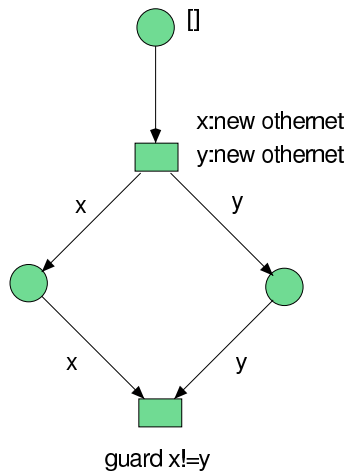
Figure 3.10: The net `creator`   Figure 3.11: The net `othernet`

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.

31

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 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. 3.8 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.

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 [2]. 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.
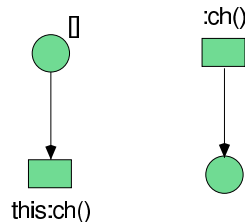
Figure 3.12: The net synchro

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.
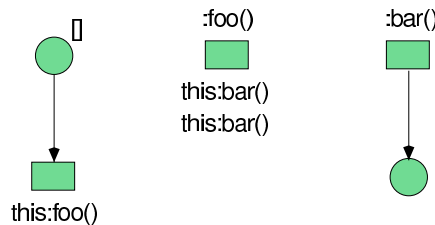
Figure 3.13: The net multi

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.
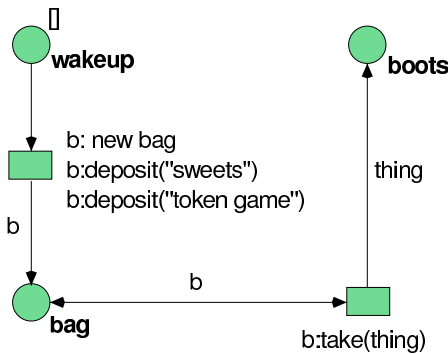
Figure 3.15: The net `santa`   Figure 3.16: The net `bag`
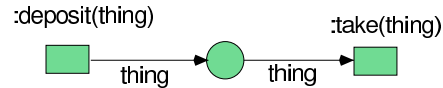
It is possible to create synchronization loops where the invocation of a channel results in the invocation of the same channel. This should be avoided, because it might throw the simulator into an infinite loop. A different search strategy could have avoided this problem, but it would have incurred a significant performance cost.

We already established that it is not desirable to use channels to implement control structures. Hence need for cyclic channel invocations is very rare. If you want to guard yourself against bad use of channels, run a channel check on the finished model. This is available in the simulation menu and described in Subsection 4.4.5. Of course, the check cannot protect you from infinite loops that occur in invoked Java methods, but it rules out loops due to channels.

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.

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 [11]. A workflow system of a law enforcement agency is the basis for the nets in `samples/prosecute`. They are based on the article [14], where this example is attributed to W.M.P. van der Aalst.

3.8 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.simulator.NetInstance`, and they implement only the methods of `de.renew.simulator.NetInstance`.

3.8.1 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*.
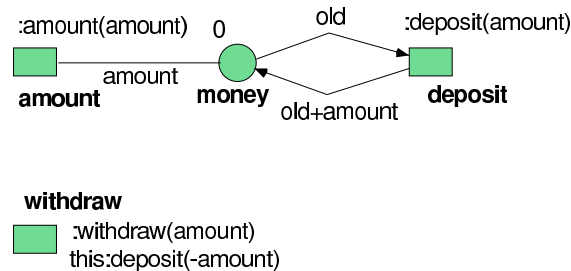
Figure 3.17: The net `account`

The net from Fig. 3.17 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`, because `samples.call` seems to be the proper package. 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);
  }
}
```

35

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`. For other operating systems we do not currently supply a convenient shell script, but you can achieve the same effect by running

```
java de.renew.call.StubCompiler samples/call/Account.stub
```

or similar commands. Now

```
javac samples/call/Account.java
```

compiles the Java source resulting in the file `Account.class`. We will now use this class inside a reference net, but it could be used in Java code just as well. 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. 3.18 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.
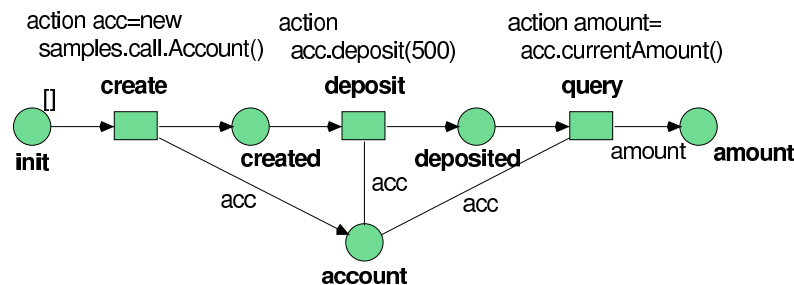
Figure 3.18: The net `customer`

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:

```
...
(2) Synchronously
(2)     Testing account[1] in customer[1].account
(2)     Removing [] from customer[1].created
(2)   Firing customer[1].deposit
(3) Synchronously
(3)     Removing int(0) from account[1].money
(3)   Firing account[1].deposit
```

36

```
(3)      Putting int(500) into account[1].money
(2)      Putting [] into customer[1].deposited
...
```

The transition `deposit` of `customer` fires at step (2), 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 (3). 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 and output arcs 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.

3.8.2   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 [15].) 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. 3.19 and 3.20.
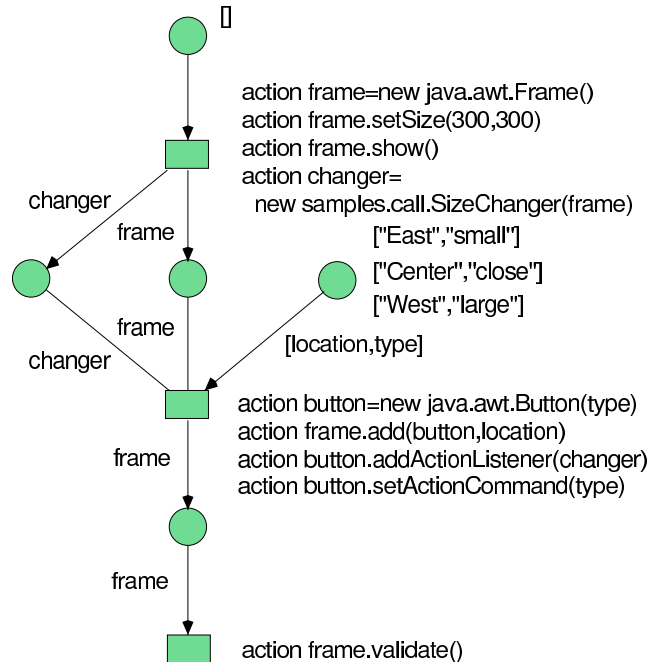
Figure 3.19: The net **buttonmaker**

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 uncloseable windows might accumulate on the desktop.

3.8.3   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.
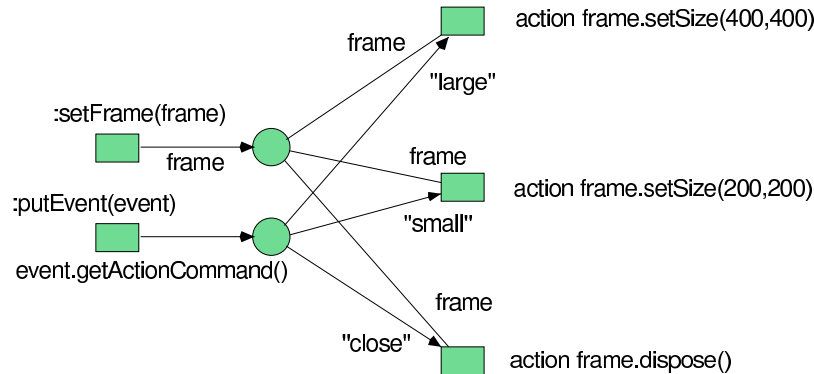
Figure 3.20: The net `sizechanger`

```
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 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 3.8.1, i.e., by calling

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

or equivalent commands.
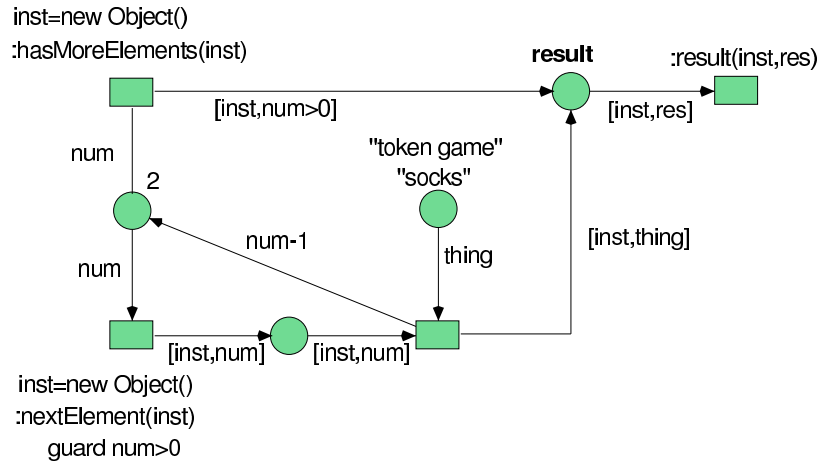
Figure 3.21: The net `enumbag`

Fig. 3.21 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. 3.22 you can see Santa distributing the Christmas presents.
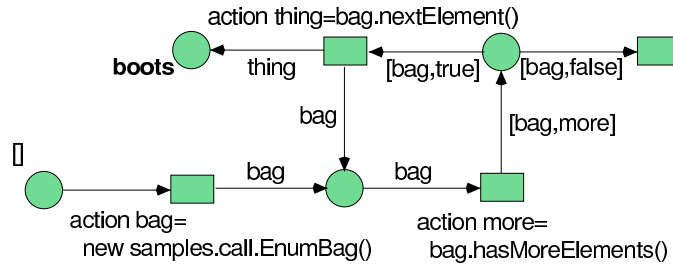
Figure 3.22: The net `enumsanta`

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.

3.9 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 modelling 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 where made in the simulation engine to allow their implementation, so that robustness of Renew is not affected.

3.9.1 Flexible Arcs

Flexible input arcs and flexible output arcs were introduced by Reisig in [12]. 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.

Renew 1.2 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 were preferred over vectors, 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 these cases we suggest that you use helper methods written in Java to put the flexible arc's tokens into an array.

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 in the case 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.
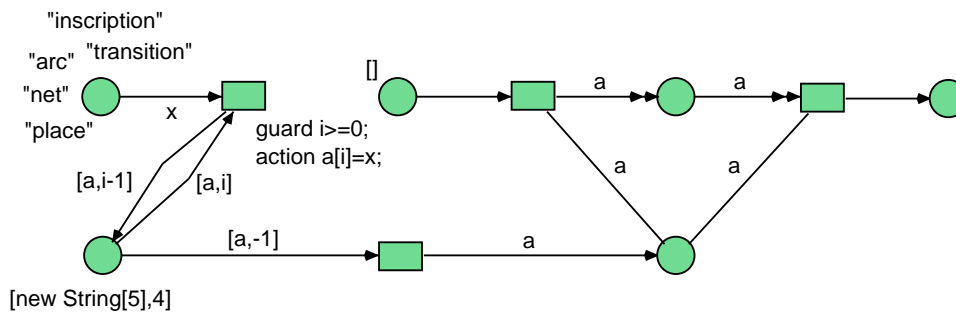
Figure 3.23: The net `flexible`

The net from Fig. 3.23, 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 syntactical or semantical variant proves superior.

In the book [13] 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.
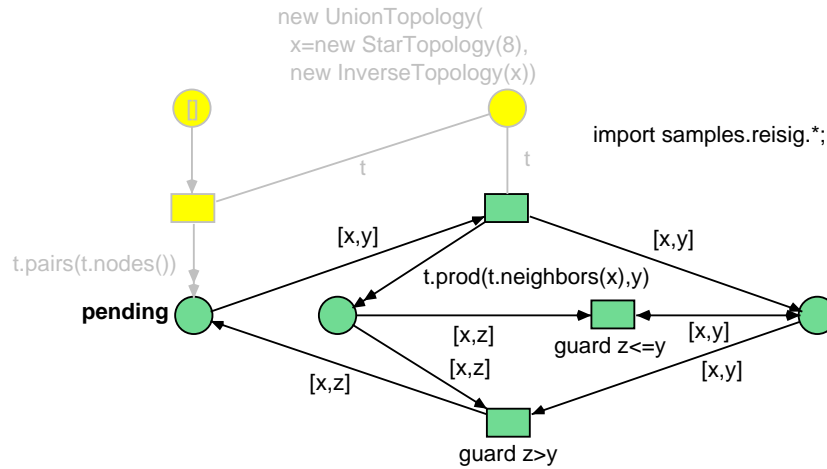
Figure 3.24: The net `election`

The net from Fig. 3.24 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 [13]. 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. 3.25 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 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.
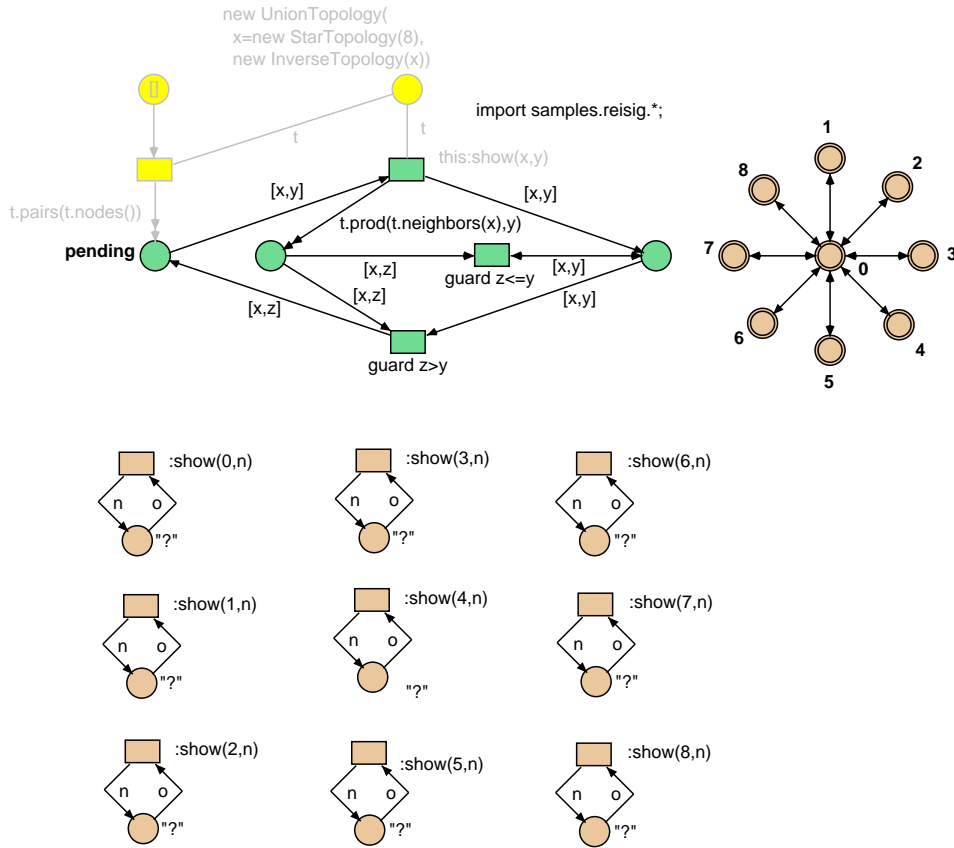
Figure 3.25: The net `visualelection`

3.9.2 Clear Arcs

Renew 1.2 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. [9] 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 2.6.6.

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. 3.26 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,

43

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 arc from the place `waiting` to the transition `perform`. This way, the juggler only starts his business when there is at least one spectator.
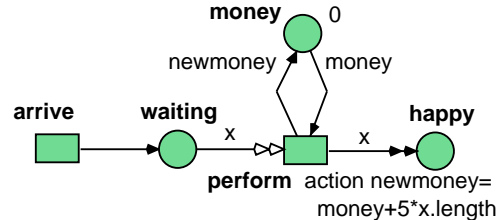
Figure 3.26: The net `juggler`

3.9.3 Inhibitor Arcs

^{Renew}
1.2
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 2.6.6.

Some varieties of inhibitor arcs were suggested in the literature, see [3] and [9] for a recent approach and [1] for a consideration of concurrency issues. The papers also give further references. The formalism presented here is less general than that presented in [3], 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 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.

In Fig. 3.27 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 2.6.6.
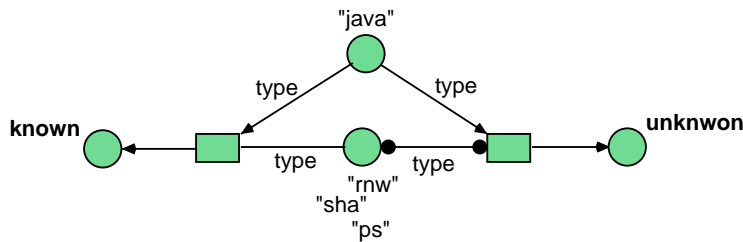
Figure 3.27: The net `filetypes`

3.10 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.

Renew 1.3 A timed simulation mode was added to Renew. To this end, 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.

Timed nets can *only* be simulated in the sequential simulation mode as described in subsection 2.6.6, which is typically started with the command `srenew`. The ordinary simulation mode will not be able to handle timed nets and it will issues an error message at the start of the simulation.

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 move after `t` time units. Input arcs can require that a token remains available for 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, altough 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. Double 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.

In Fig. 3.28 you can see an example model of a sea port where ships arrive and are unloaded. The loading the ships is not displayed. At the upper righthand corner, you can see that new ships are created, each ship being numbered accordingly. Probabilistically it is determined whether the ship need 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 afterwards. If a declarations 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.
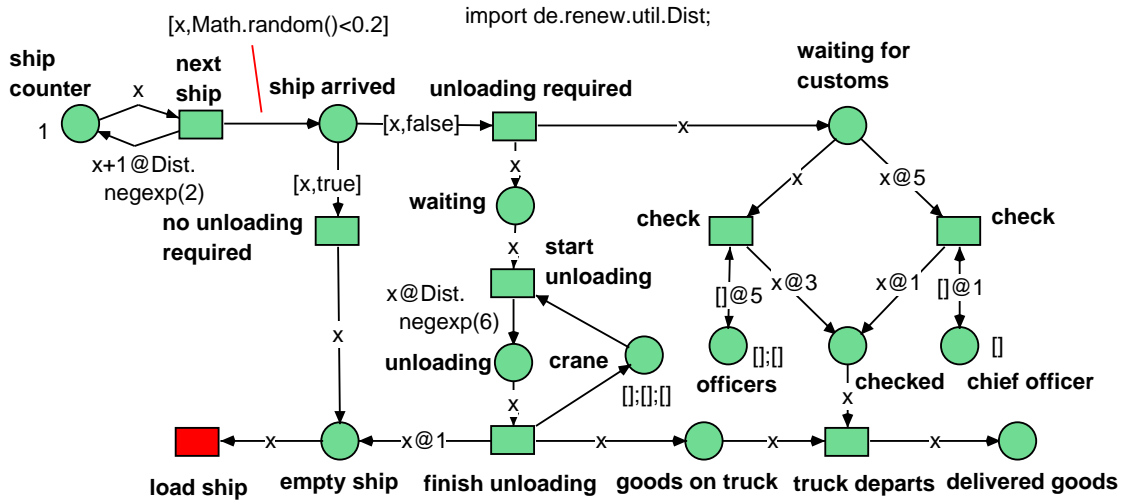
Figure 3.28: The net `port`

Note how we use a reserve arc to model a resource that it 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.

3.11 Pitfalls

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

3.11.1 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 modelling errors.
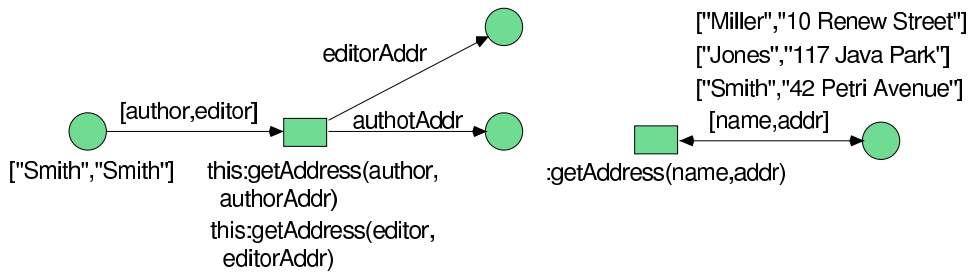
Figure 3.29: The net `reserve`

The net from Fig. 3.29, 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.

46

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.

3.11.2 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 backwards. 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. 3.29 contains such an error, because in one place `authorAddr` is misspelled as `authotAddr`.

3.11.3 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.
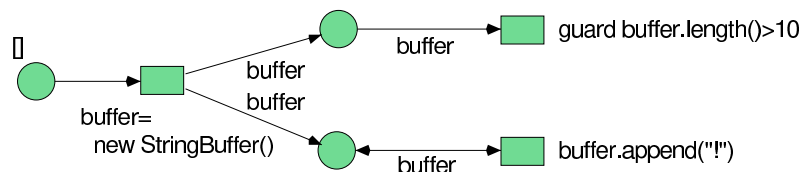
Figure 3.30: The net `buffer`

In Fig. 3.30 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. Afterwards it is not rechecked, because its input place did not change its marking.

You will also notice that the length of the string increases by 2 during each cycle. How can that be? The call to `append` is not contained in an action, so that it is evaluated once during the search for a binding and once during the actual firing. Note that this is a relatively harmless scenario.

47

3.11.4   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 2.6.7.

3.11.5   Net Stubs

A net stub 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. Therefore you must load every net that might be required during the simulation into the net editor at startup time.

There is no dynamic loading of nets similar to the dynamic loading of classes in Java. We consider to add such a feature in future versions of Renew, but at the moment you will simply get an error message, if you use an unloaded net at runtime.

There is no simple possibility to check the validity of net names at compile time, so this error cannot be detected early either.

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 [5]. The basic drawing capabilities are mainly taken over from JHotDraw, while the PostScript output, 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 Mouse handling

Most current computer platforms use a mouse with at least two mouse buttons. Whenever this manual instructs you to do mouse operations, you should use the *left* mouse button. You should use the *right* mouse button only when it is especially indicated to do so.

If your mouse has three or more buttons, the excess buttons usually behave like the right mouse button.

Your operating system might support a special option to switch the two mouse button, so that left handers can operate the device more easily. This will also switch the meanings of the mouse buttons within Renew.

Older Apple Macintosh mice have got one mouse button only. In this case, you can press and hold the Apple key while clicking the single mouse button, whenever this manual commands you to press the right mouse button. On other operating systems, too, you might be able to use a single button mouse by pressing some control key whenever you need access to the right mouse button.

In all cases, you can substitute right mouse clicks by appropriate menu commands or tool selections, but the right mouse button greatly adds to drawing speed.

4.2 Basic Concepts

When working with Renew, you edit so-called drawings. A drawing consists of many drawing elements, called figures. Each drawing is displayed in a separate drawing window. Since you are expected to work on many different drawings and thus have many different windows open at the same time, it would consume lots of valuable desktop space to repeat a menu bar and tool buttons in every window. To avoid this, all commands have been grouped into one central window, the Renew window, which contains a menubar, a toolbar and a status line (see figure 4.1). This might seen a bit unfamiliar for Mac users, but is related with the platform independence of Java.

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,
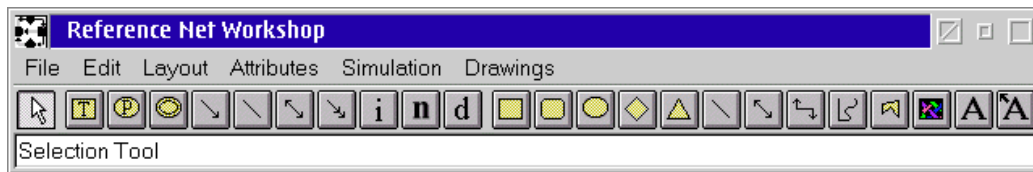
Figure 4.1: The Renew Window

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 4.4. 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 4.3. 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 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 using a program icon you created, depending on your operation system. When Renew is started without any parameters, a new, empty drawing window is opened.

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.

4.3 Tools

In the toolbar, several tool buttons are displayed, which can be selected by clicking on them. At any point in time, exactly one tool 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. But if you select `Toggle Sticky Tools` from the `Edit` menu, the current tools will remain active until you explicitly select another tool.

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.

4.3.1 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:

50

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 depressed 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 depressed 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 4.3.2: 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 depressed 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 4.3.2: The Connected Text Tool.

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 4.3.2 in the subsection of the corresponding figure's tool.

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 4.4.5.

4.3.2 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 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

type of figure	double click	right click
rectangle, ellipse, . . .	select	select child
text	select	text edit
attached text	select parent	text edit
transition	select	attach inscription `:s()`
place	select	attach inscription `[]`
virtual place	select associated place	attach inscription `[]`
arc	select	attach inscription `x`
declaration	select	text edit
inscription	select parent	text edit
name	select parent	text edit
transition instance	open binding window	fire arbitrary binding
simple marking	open marking window	show expanded marking
expanded marking	inspect token	show simple marking

Table 4.1: Summary of selection tool operations

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 4.3.1). 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.

All newly created figures have a black outline and cyan as the fill color (if there is any). To change these attributes, use the `Attributes` menu (see Section 4.4.4).

> To create figures with the same attributes as an existing figure, use copy & paste (see Section 4.4.2).

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 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 afterwards.

   A line figure has no fill color, but it respects the pen color (see Section 4.4.4).

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* in a connection. Activate the connection 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. 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). With this feature, you can produce connections which do not go straight from one figure to another but have some intermediate "pin-points". If you want to get rid of a pin-point, simply select the connection and right-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 the 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

The scribble tool lets you scribble in a drawing with your mouse, just like the famous Java applet. More precisely, a scribble figure traces the mouse movement while the button is held down and thus defines several points, which are connected by lines. You can also define single points by single mouse clicks. The creation mode is ended by double-clicking at the last point. The clou about the scribble figure: After it has been created, every single point can still be changed by dragging the corresponding white, square handle. To drag the whole figure, start dragging on a line segment rather than inside a handle, or deselect the figure first and then start dragging anywhere on a line 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 new function is that 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 new 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 Image Tool

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

Be aware that the PostScript output does not (yet) support transparent GIF images, but only the normal print function does. Also, there seems to be a problem with bitmap images that have an odd (in contrast to even!) height: the last pixel line is omitted in PostScript output.

After you confirmed the file selection, the dialogue 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.

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 one line 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. It is important to note that you do not necessarily see all of the text while editing. This does not mean that the text is deleted. It is just that the size of

the text editor is not adapted dynamically. 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.

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 4.4.4). On some systems, it may take a while until the fonts for different sizes are loaded the first time, so don't get out of patience.

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, this time in a more appropriate size. 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 4.3.3: 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.

4.3.3 Net Drawing Tools

Now it is really getting interesting: This group of tools allows you to draw Petri nets that have a semantical 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 syntactical 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 afterwards, 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 semantical meaning to the simulator. A rectangle figure is ignored by the net execution engine.

- To create a transition with a default size (which cannot be customized at the moment), 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 4.3.3: 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 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

Renew 1.1 This tool has been added in Renew 1.1 to improve the readability and graphical appearance of nets in which certain places are used by many transitions.

The virtual place tool is used to create virtual copies of a place. Other Petri net tools sometimes call such a virtual copy of a place a fusion place. If the contents of a place is needed for many transitions, the readability of the net decreases because of many crossing arcs. With a virtual place copy, you can draw the same place many times, thus avoiding such crossing arcs and arcs over long distances.

You create a virtual copy of a place by activating the virtual place tool, then clicking on the place you want to copy (this can also be a virtual place!) and keeping the mouse button

down while dragging the virtual place figure to its destination location. The virtual place can be distinguished from a normal place by the double border line (see the graphics inside the tool button). To find out which place a virtual place belongs to, just double-click the virtual place. To make this relation visible in printed versions of your nets, you should copy the name of the place to the virtual place. Unfortunately, the tool does not take care of the names of virtual places automatically. Another solution supported by the tool is to give each group of a place and all its virtual copies a different fill or pen color. All places belonging together will change their colors if you change the color for one place.

During simulation, every virtual copy of a place contains exactly the same token multiset as its original place. Still, it is possible to determine separately for each virtual place (and the place itself), whether the marking should be displayed as cardinality or expanded (see Section 4.4.5).

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 above). The differences are, like above, that an arc has a semantical 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 4.3.3: 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 3.11.1). 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 3.11.1.

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 3.9.1.

There are two additional arc tools that are only present in the extended sequential mode as described in Subsection 2.6.6:

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 3.9.2.

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 it end points. Again, the direction does not matter. For the semantics of inhibitor arcs, see Section 3.9.3.

57

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.

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 4.3.2: 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 semantical meaning to the simulator. By default, inscriptions are set in plain style, while labels (text without semantical 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 console window exactly like you specified in the name figure. Place names are used in the log window 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 4.4.5).

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 semantical 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.

4.4   Menu commands

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

4.4.1 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 Drawing

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 4.4.6).

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

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 console window and in the status line of the Renew window. Otherwise, the drawing is added to the list of drawings in memory (see Section 4.4.6) and opened in a new drawing window. The keyboard shortcut for this function is `Ctrl+O`.

Save Drawing

This functions saves the active drawing (see Section 4.2) 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 functions 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.

Save All Drawings

This function saves all drawings that are currently in memory (see Section 4.4.6). 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 dialogs, 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 4.4.6).

Renew
1.1
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.

Set Selection as Icon

Renew
1.2
This new feature allows you to assign icons to your nets. These icons will be displayed during simulation, whenever a token is displayed in `expanded` mode (see subsection 4.4.4) 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 4.2 and 4.3 show modified versions of the nets from the Santa Claus example.
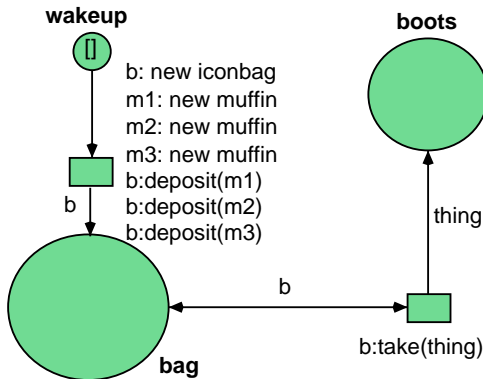
Figure 4.2: The net `iconsanta`   Figure 4.3: The net `iconbag`
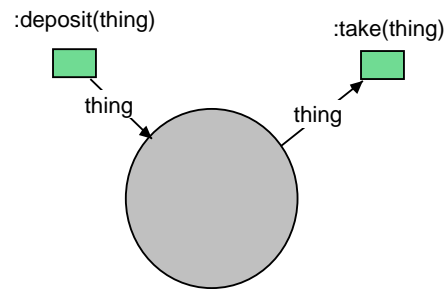
Add an icon to the `bag` net by drawing an ellipse, coloring it grey, and drawing a polygon which looks like the closure of the bag. Add a text with the string `BAG $ID` to the drawing. Be careful not to connect the text to any other figure (see bug below). Group together all new figures (`Edit | Group`). 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 favourite 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 4.4. 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.
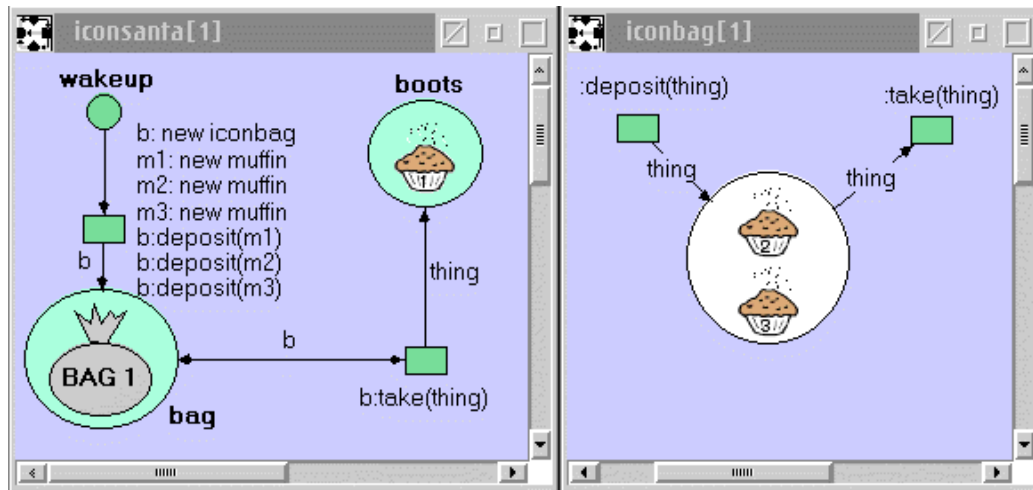
Figure 4.4: The Santa Claus Example with Icons During Simulation.

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. This is why we implemented a PostScript output feature (see next section).

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

Export PostScript...

This function produces PostScript code that renders the current drawing. You have to select the file (default extension `.ps` in which to store the PostScript code. Using PostScript has the main advantage that the resolution and quality is much higher than that of the standard Java printing facility (see previous section).

Export EPS...

If you want to include net drawings into written material, you should use an Encapsulated Postscript (EPS) file instead of an ordinary Postscript file. The EPS file can be used to insert graphics into other documents, e.g. in LaTeX, StarOffice, MS Word, and others.

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

For developers: The class `de.renew.util.PostScriptWriter` (which is a subclass of `java.awt.Graphics` that generates PostScript code) can easily be used in other applications. If you want to do so, please contact Frank Wienberg (see Section A) to find out about capabilities and limitations of this class.

61

Export XML...

This menu entry is only available after you have installed an XML parser as described in section 2.1.

The current drawing is exported in a experimental XML file format. This function serves mainly for illustrative purposes during the discussion of a common XML Petri net description. We expect that the common XML file format will ultimately emerge, building on the ideas that were discussed at the XML meeting during the ICATPN'2000 [16].

If you would like to experiment with this format, you should start with the document type definition file `de/renew/gui/xml/xrn1.dtd` which contains a description of this file format. Also, there aformentioned web site [16] references a paper on this file format.

This file format is subject to change without notice! In fact, do not even expect to be able to read your XML files with the next version of Renew! Always keep one saved version in the basic Renew format! Note that not all information present in a net drawing is present in the XML file!

Export to Macao...

Exports the active drawing in the Macao format. Macao (see [8]) is a powerful Petri net simulation and analysis tool. This function is probably of use for a very limited number of people and still in beta status. Do not expect every Renew net to be mappable to Macao. This function can merely be used to check the net structure of a single net. Also, you may have to manually adjust inscriptions etc.

Export to Woflan...

Exports the active drawing in the Woflan format. Woflan (see [4]) 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 modelling errors in their structure.

Export Shadow Net System...

Renew 1.1 Exports the *shadow net system* generated by all open drawings, which can be used by the shadow simulator (or other tools). Sometimes it is useful to simulate a net standalone without starting the graphical user interface. This can be achieved by exporting the *shadows* of the net elements to a file. Only the semantical information is contained in the shadows, but not the visual appearance.

For example, load the file `gcdtyped.rnw` from the directory `samples/simple` as the only drawing. Invoke this menu item and select `gcdtyped.sha` as the target file. Exit Renew. To start a standalone simulation, use

```
java de.renew.application.ShadowSimulator \
  de.renew.simulator.ConcurrentSimulator gcdtyped.sha gcdtyped
```

assuming that your operating system allows the continuation of commands after a backslash. In other cases, enter the whole command in one line, but without the backslash. The simulator should fire transitions until deadlock. The concurrent simulator cannot automatically detect the deadlock, because background processes could theoretically reactivate some transitions. Using

```
java de.renew.application.ShadowSimulator \
  de.renew.simulator.SequentialSimulator gcdtyped.sha gcdtyped
```

you can start the sequential simulator, which automatically terminates upon a deadlock. However, the sequential simulator cannot be used if you want to call nets from Java as described in Section 3.8.

Before exporting a collection of nets to the shadow simulator, it is recommended to do a syntax check on the net. In any case, the syntax errors will be detected before the start of the simulation.

In Subsection 4.4.4 we discuss a useful feature that switches off the trace messages about firing transitions. After a net has been sufficiently debugged, these messages become useless and possibly even annoying. Of course the Java expressions associated to a Net are still being executed even after the trace messages have been disabled.

Import XML...

Here you can import an XML file that was created by Renew as described in subsection 4.4.1. Whenever possible, the graphical and semantical information is restored from the file.

Note again that you will not be able to import an XML file of a different Renew version with this command!

Import Shadow Net System...

Renew 1.2 This menu lets you import a previously (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.

Renew 1.2 We now offer an automatic graph layout function that can ease the task of making an imported net readable. See Subsection 4.4.3 for details.

Exit

Tells Renew to terminate.

Renew 1.1 All drawings are closed as if you closed them manually, which means that now Renew asks you about saving changed drawings (see Subsection 4.4.1).

4.4.2 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.

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 4.2). 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.

Connection figures should only be copied and pasted together with the figures they connect. Otherwise, pasting will create a copy of the connection figure that is not associated correctly to its start and end figure. To correct this, drag both connection handles and release them on other or even on the same figures. The same holds for connected text figures. After pasting, a connected text figure has to be assigned to a new parent like described in Section 4.3.2: The Connected Text Tool.

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. Mind the bug described above for the paste function!

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).

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 makes 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

Renew 1.2 Commands that allow selection or deselection of large sets of nodes were added. They allow the user to select groups of logically related net elements together. For selecting locally related net elements or individual net elements see Subsection 4.3.1.

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.

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 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.

64

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 4.3.1, but to restrict the selection to a certain type of figures afterwards.

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.

Toggle Sticky Tools

Selecting this menu toggles the Sticky Tools mode of Renew. By default, a tool will deactivate itself after the completion of a single task. Afterwards, the selection tool will be reenabled. In Sticky Tools mode all tools will remain activated until you choose another tool explicitly.

In general, Sticky Tools mode is most useful during the initial creation of nets and the default mode is more apt to later modification stages. But of course, which mode to use also depends on your personal preferences.

4.4.3 Layout

^{Renew}
1.2 This menu has been renamed from `Align` to `Layout`, since it now includes some layout features beyond aligning. The new `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 function is also very basic, because the grid density is not customizable.

Lefts, Centers, Rights

These functions all align the selected figure's x-coordinates, i.e. moves them horizontally. The figure selected first is the reference figure which determines the x-coordinate for all others. `Lefts` sets the left-hand side of all selected figures to this x-coordinate, `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.

Tops, Middles, Bottoms

These functions work exactly like the ones in the previous section, except that the y-coordinate is changed. Thus, figures are moved vertically in order to be aligned with their tops, middles, or bottoms.

Send to Back, Bring to Front

Renew 1.2 These menues have been moved here from the `Edit` menu.

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 4.3.1).

Automatic Net Layout...

Renew 1.2 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 strucutre of the net.

This menu entry start an automatic net layout on the current drawing. While this mode is active, the nodes of the net are moved according to certain rules that are to some extend inspired by physical laws 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 oritented. The torque works towards 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 or the physical model using sliders. The first slider controls the length of the springs. Some diagrams tend to clump together too much, which might can 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 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 towards to 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 equilibirium 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 towards 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.

4.4.4 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.

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.

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).

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`. If the figure is an arc, its semantics are changed accordingly.

Font

Only applicable to text-based figures, this attribute sets the font for the complete text of this text figure. Not all fonts are available on all platforms. It is not possible to use several fonts inside one text figure (but still, this is a graph editor, not a word processor or DTP application).

Font Size

Only for text-based figures, select one of the predefined font sizes given in point with this menu.

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 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.

Text Type

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

If a text figure is a `Label`, it has no semantical meaning at all. If it is a `Inscription`, it is used for the simulation (see Section 4.3.3: The Inscription Tool). A `Name` text type does not change the simulation, but makes the log more readable (see Section 4.3.3: 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 4.3.2: The Connected Text Tool) and changing the duplicate's text type to `Name`.

Trace

This menu and the next one really refer to a net simulation. However, they have been placed under the `Attributes` menu, because both refer to attributes each single net element can have.

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

- If a transition's trace flag is switched off, no firing of this transition is printed to the log window.

- A place's trace flag has no effect on the simulation output for reference nets (but might be useful later on).

- If an arc's trace flag is switched off, the lines informing about tokens flowing through this arc are omitted.

Marking

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

In version 1.1, there were two ways to display the marking of a place during simulation: Either just as the number of tokens (`cardinality`), or as the verbose multiset of tokens (`expanded`). These modes could be switched for each place individually, but only during run-time, and each simulation started with all places in `cardinality` mode.

Renew
1.2 In version 1.2, we have added a third mode: `highlight only` does not display any token information, but just highlghts the place during simulation, whenever it contains *any* tokens. The `Marking` menu was added to have better control over the display mode.

Renew now stores for each place the preferred display mode chosen by this 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

Renew
1.3 Using this attribute, you can request breakpoints for certain places and transition. 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.

Attributed breakpoints, too, will show up in the breakpoint menu. Please see subsection 4.4.7 for a detailed description of the possible breakpoints. Note that you can set at most one breakpoint for each net element using the attribute menu.

4.4.5 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 Section 4.4.6). The drawing window containing the net that is to be instantiated initially has to be activated.

Refer to Section 4.5, 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 4.3.3: 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`.

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`.

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 3.8), 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.

Renew
1.1 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`.

Save simulation state ...

Renew
1.2 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 afterwards.

- 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 neccessary 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 4.5 for these nets, but the menu commands `Simulation step` and `Run simulation`
will still work and trace messages will still be printed. 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 will be printed to the console. An instance drawing will
still be opened, but it will not neccessarily 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 2.6.7), 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.

Syntax Check Only

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.

Channel Check

This menu entry checks for patterns in the synchronous channel inscriptions that
could throw the simulator into an infinite loop. In these cases, one or more transi-
tions have a downlink and an uplink, so that a transition can invoke itself through
synchronous channels. There are occasions when this is sensible, but it should only
be used in experimental models.
For productions models you should make sure that the simulator behaves correctly
by running this check once. If the check fails, it reports the cyclic dependency of
channels that was found. Removing this problem can cause a substantial amount
of work, because a loop needs to be made explicit in the net structure.
The check can also fail if a downlink does not have a matching uplink. That means
the offending transition can never fire. This is usually a misspelled channel name
and not a deep problem.
It is not reported as an error when an uplink does not have a matching downlink,
because this simply indicates that some functionality of a net is not yet needed and

it might well be required in the future. The unmatched uplink might also be needed for a Java stub as described in Section 3.8.

Name Check

Renew 1.1 This menu entry checks for net elements with the same name within one net. This error is not reported in the normal syntax check, because there might be occasions when it is sensible to have a place and a transition that share the same name, or even two transitions with the same name. But since a repeated name can indicate a subtle modelling problem, this check is provided.

Associate Highlight

Renew 1.2 The simulation feedback has become much more elaborated in this version of Renew. Now it is not only possible to select the kind of feedback given for the marking of a place (see Subsection 4.5.1), 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 behaviour, 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.[1] 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 (displayed in a brighter colour) exactly when the net element is highlighted.

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 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 cancelled.

4.4.6 Drawings

This menu contains a list of all drawings loaded into memory. A drawing can be loaded supplying its file name to Renew as a command line argument, invoking the `Open Drawing...` menu, or created through the `New Drawing` menu. A newly created drawing can be named and any drawing can be renamed by saving it using the `Save Drawing as...` menu.

By selecting a drawing in the `Drawing` menu, its window is raised and becomes the active drawing window. In the menu, the name of the active drawing appears checked.

---

[1]It is even possible to associate another net element as a highlight, but this is not recommended, as it can lead to confusion.

Unfortunately, a window that was "iconized" (called "minimized" on some platforms) by the user can only be re-opened by the user, which seems to be a Java bug. Another Java bug is that under Unix, some window managers refuse to display titles in Java windows. As a workaround, the `Drawings` menu becomes even more important, as it is the only way to find out the names of all your nets.

4.4.7 Breakpoints

^{Renew}
1.3 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 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 loose 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.

4.5 Net Simulations

During simulation, there is textual and graphical feed-back. The Java console prints a log of the simulation that is briefly described in Section 4.4.5. In this log, you can see exactly which transitions fired and which tokens were consumed and produced. But you can also view the state of the various net instances graphically and you can influence the simulation run.

4.5.1 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 such a marking, another window appears, containing detailed information about the tokens.

Renew
1.1 You can now 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 1.0. To switch between the simple (cardinality of the multiset) and the expanded 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.

4.5.2 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

the value itself. The token value is usually displayed as some String, using the Java method `toString()`, which is supported by every Object.

There is a special function to gain access to other net instances. If a token's value is a net instance, you can double-click it in the current marking window. Then, a new net instance window for that net instance is opened or the corresponding net instance window is activated, if it already existed.

Renew
1.1 This also works for net references contained within a tuple, or even within a nested tuple. Just double-click on any tuple component which is a net instance.

We also fixed the bug that when you closed a net instance window, all current marking windows corresponding to place instances of that window would not react to changes in the marking any more. Now, 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.

A really new feature is that when you double-click a token which is a reference to some `java.awt.Window` object (like a `java.awt.Frame`), this window is raised to the top.

4.5.3   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 is only makes sense when the simulation is performed step by step (see below).

Renew
1.1 Interactive simulation is now possible. There are two ways to force a specific enabled transition to fire:

- 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!

75

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 more thorough 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 labelled `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 4.3.2: 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.

4.6.1   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.

4.6.2   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 postfixed 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.simulator.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.

4.6.3 Late Error Messages

Here we discuss the error messages that are not reported during the immediate check, but only during the complete check before the simulation.

Cannot clear untyped place using typed variable

An 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?

Expression of array type expected

An inscription of a flexible arc is expected to be either completely untyped or typed with an array type. It is not allowed to use enumerations or vectors for flexible arcs. Although these would constitute a means to denote multisets of tokens, they are not permitted due to type security. 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.

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?

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 problem. 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 simulation mode cannot handle such inscriptions, which require the sequential net formalism. Maybe you want to save the net and reopen it in the sequential simulation mode as described in subsection 2.6.6?

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?

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?

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 Channel Check Messages

These errors are determined during an explicit channel check.

Channel ... can invoke ... This completes a cycle

A cyclic channel dependency was detected. The offending channels are listed. Often the current channel structure was intended, but it was not obvious that this might lead to an infinite loop. Try to restructure your model, if it is supposed to be used in production applications. A cyclic channel invocation does not necessarily lead to unusable models. In any case reconsider your model carefully.

Channel ... can invoke itself

See the previous point for details. However, in this case there is no doubt that the loop was created intentionally.

Channel ... cannot be satisfied

For some downlink the syntax checker could not find an uplink with the same name and the same number of arguments. Maybe a channel name is misspelled? Maybe a parameter in a channel inscription is missing? Maybe the uplink is provided in a net that is not yet loaded?

4.6.5 Name Check Messages

This error is determined during an explicit name check.

Detected two net elements with the same name

Not even a place and a transition are supposed to have the same name during this check. Maybe some net elements are named identically because they do the same thing, although at different times or in different situations? Make the names more specific. You may include spaces in the names. Maybe you copied a set of net elements and did not change the names yet?

Chapter 5

Customizing Renew

Because Renew is available with source, it is tempting to create a customized version of it. This is indeed possible, but there are a few drawbacks:

- Currently Renew is only partially commented. Especially, the comments are not yet in JavaDoc format.

- You diverge from the main development line. The internal structure of Renew might change significantly in future versions. If you send your additions/improvements to us (see Appendix A), we will try to include them in the main release.

- An internal programming guide is not available at this time.

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

5.1 Modifications and Additions

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

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

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

5.2 Inscription Grammars

If you want to customize the inscription grammar, you should download JavaCC, the Java Compiler Compiler, from

`http://www.sun.com/suntest/products/JavaCC/`

where we suggest to use version 1.1 for compatibility.

We consider doing a complete rewrite of the grammar files in order to free them of the license restrictions that currently exist. In that course of work, it might be sensible to move from JavaCC (a great program) to ANTLR (a great program that is also free and available with source).

Bibliography

[1] 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, Gjern, Denmark*, pages 182–205. 1991.

[2] Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with channels for synchronous communication. Technical Report DAIMI PB–390, Aarhus University, 1992.

[3] 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`.

[4] Eindhoven University of Technology. *Woflan – The Workflow Analyser*, 1998.
WWW page at `http://www.win.tue.nl/~woflan/`.

[5] Erich Gamma. *JHotDraw*, 1998.
Available at `http://members.pingnet.ch/gamma/JHD-5.1.zip`.

[6] 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.

[7] 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-Universitt, Institut für Wirtschaftinformatik, Frankfurt am Main, Fachbereich Informatik, October 1999.

[8] Laboratoire d'Informatique de Paris 6. *Macao Graph Editor Framework*, 1998.
WWW page at `http://www-src.lip6.fr/logiciels/macao/`.

[9] 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`.

[10] Doug Lea. *Overview of the `collections` Package, Version 0.96b*. State University of New York at Oswego, 1998.
WWW page at `http://gee.cs.oswego.edu/dl/classes/collections/`.

[11] 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.

[12] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1–2):1–34, 1991.

[13] Wolfgang Reisig. *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets*. Springer-Verlag, 1998.

[14] 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.

[15] Peter van der Linden. *Just Java*. The Sunsoft Press Java Series. Prentice Hall, 1996.

[16] Proceedings on the meeting on XML/SGML based interchange formats for Petri nets, 2000.

WWW page at `http://www.daimi.au.dk/pn2000/Interchange/`.

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 TGI
— Renew —
Fachbereich Informatik, Uni 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

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 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.

B.1 Contributed Parts

Renew uses several parts that were previously developed by other people and have been made publicly available.

B.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. As soon as Java 1.2 becomes widely available, we plan to follow his advice.

B.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.

B.1.3 Code Generated from JavaCC

Some of the code of Renew was generated by the parser generator JavaCC. The relevant license information states:

> 3. DEVELOPED PRODUCTS
> You may use the Software to generate software program(s) ("Developed Programs"). Sun claims no rights in or to the Developed Programs.

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.

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.

B.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`) 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.

B.1.5 Graph Layout Algorithm

The graph layout algorithm used in the class `de.renew.util.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.

B.1.6 The SAX API

The package `org.xml.sax` was created by David Megginson. The relevant license information states:

No warranty; no copyright – use this as you will.

B.2   Original Parts

This copyright section deals with those part 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.

B.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.

B.2.2   Java Source Code and Executables

Sources and executables are copyright 1998 by Olaf Kummer and Frank Wienberg. You can distribute these files under the GNU General Public License.

You should have received a copy of the GNU 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.

B.3   Created Parts

You are permitted to use works that you create with Renew (i.e., Java stubs, net drawings, PostScript output, simulation states, and other exported data) without restrictions.

B.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.

B.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 `http://www.opensource.org/osd.html`), please contact the authors.