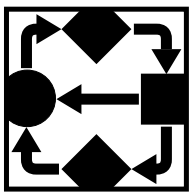
Renew – User Guide

Olaf Kummer
Frank Wienberg

University of Hamburg
Department for Informatics
Theoretical Foundations Group
Distributed Systems Group

Release 1.0
March 5, 1999



Contents

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),
- drawing the nets (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. [7] 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 mayor 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. It is, to our knowledge, the first Petri net tool that implements bidirectional 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. For object-orientation this is very helpful. Having object-orientation without garbage collection is like having a fireplace without a chimney: it is hot, but smoke will prevent your sight.

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

- There are currently no 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. You can dynamically explore the state of the simulation, but there are no statically checkable properties.

  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 influence decisions where non-determinism is involved, but the simulation engine takes care of solving these conflicts randomly. Interactive parts have to be explicitly implemented using Java inscriptions. In our formalism, there is also no notion of simulation time or firing probabilities. All these are features that may be added later on, possibly as third-party contributions, which should be fairly easy because of the open architecture of Renew.
- 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 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.

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/macos/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 is JDK 1.1.6 or 1.1.7, depending on the operating system.

2.2 Possible Collisions

While Renew is based on the `collections` package by Dough Lea [5] and the `JHotDraw` package by Gamma [3], 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 Installing Renew

Renew is distributed in the form of two jar-files, namely renew1.0base.jar and renew1.0source.jar. While the former contains all files that are required for the operation of Renew, the latter includes the sources files, which are only needed if you intend to modify Renew. For platforms where jar is not available, we also provide zip-files.

The base Renew distribution consumes about 3 MByte. The source distribution adds another 4 MByte to that figure.

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 renew1.0base.jar
```

or

```
unzip renew1.0base.zip
```

if your Java environment does not support the jar command.

A directory renew1.0 will be created in the current directory. The subdirectory renew1.0/doc contains documentation files, i.e. this manual, the README, and the GNU GPL. The subdirectory renew1.0/samples contains example nets. The file renew1.0/renew.jar is an uncompressed jar-file that should be added to your class path, e.g., by saying

```
setenv CLASSPATH .:/some/where/renew1.0/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\renew1.0\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 CH.ifa.draw.cpn.CPNApplication
```

to start Renew.

To extract the source distribution, issue the command

```
jar xf renew1.0source.jar
```

which will deposit files in the directories renew1.0/CH, renew1.0/collections, renew1.0/fs, and renew1.0/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 renew1.0/renew.jar with the original class-files.

2.4 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.

2.4.1 OS/2

On an OS/2 Warp system, uncompress the archive `renew1.0base.jar` as described before. You can now change to the directory `renew1.0\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 1.0` folder should appear on your desktop, containing a program object for Renew as well as shadows of the documentation and the samples folder. 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.

2.4.2 Apple Macintosh

For Macs, we suggest that you do not start with the standard archive, but use the StuffIt compressed file `renew1.0apple.sit` instead. Uncompress that file and you have a Mac application that starts at your mouse click in the directory `renew1.0`. 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.

2.4.3 Unix

We supply a simple install script at `renew1.0/bin/unix/installrenew` that will handle the installation on most flavours of Unix. Run that script with

```
cd renew1.0/bin/unix
sh installrenew
```

and it will create the shell scripts `renew`, `compilestub`, and `makestub` in the same directory. Calling the script `renew` will start the net editor.

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.

2.4.4 Windows

For Windows we provide a rudimentary startup script at `renew1.0\bin\win`. Please modify it to include the classpath according to your installation.

2.5 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.

- Java is not found.

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

- Java cannot find the class `CH.ifa.draw.cpn.CPNApplication`.

  Have you set the environment variable `CLASSPATH` correctly? Maybe it is set incorrectly in your `.login`, `CONFIG.SYS` 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.

- I cannot open the sample files.

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

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

  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.

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 [4]. 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, in essence, three 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. Also, tested tokens are released as early as possible during the firing of transition. This is important, because an extended period of time might be needed before a transition can complete its firing.

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.

In Fig. 3.1 you can see a net that uses all net elements that were mentioned so far. 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

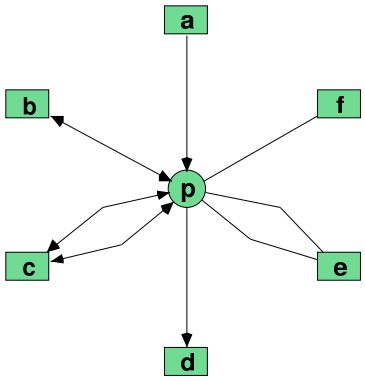


Figure 3.1: The net `elements`

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`.

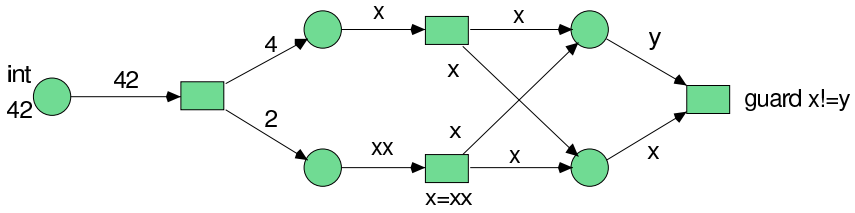


Figure 3.2: The net `coloured`

With these additions we cover the basic coloured Petri net formalism. In Fig. 3.2 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

`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

`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.

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.

All other 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 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 sizeable 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

*	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
!=	equal	primitive, reference
&	and	primitive
^	exclusive or	primitive
|	or	primitive

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

-	negate	number
~	bit complement	integral
!	not	boolean

Table 3.3: Java unary operators

usually this kind of conversion is added automatically in the places where it is needed.

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

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

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

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

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

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

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

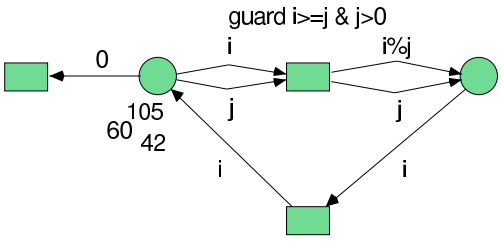


Figure 3.3: The net `gcd`

In Fig. 3.3 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 be determined only 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.

Fig. 3.4 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, declaration 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

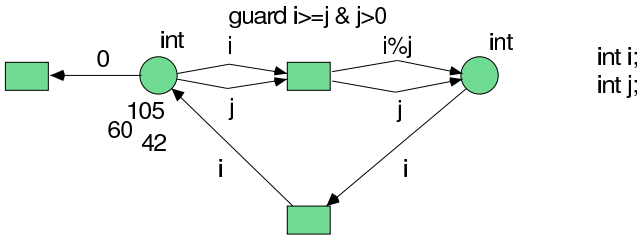


Figure 3.4: The net `gcdtyped`

cannot be a token in the place. For primitive values, we require that values of the two types can be converted into each other without losing any information. In ordinary Java `long` values are automatically converted to `float` values if that is necessary, even though such a conversion will necessarily loose information. This allows a few errors to be made, but in the context of Petri nets, where the direction of information flow may be undetermined, the problem gets worse. Hence such conversions are not done by the simulator.

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 and 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.

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.

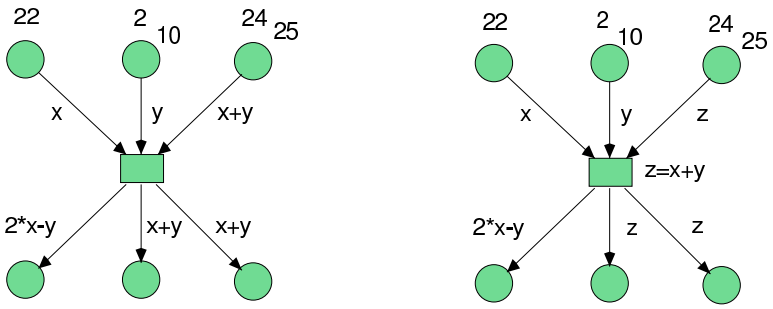


Figure 3.5: The net `equality`

In the net from Fig. 3.5 you can see two transitions that perform equivalent actions. The right transition 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=z-x` and `z=y-x`. Now the simulator can determine `y` from `x` and `z`. This is allowed, exactly because `=` does not mean an assignment but an equality specification. If a bound variable is calculated again by a redundant equation, this does not pose a problem as long as the two bindings are equal.

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

3.4.4 Method Invocations

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

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

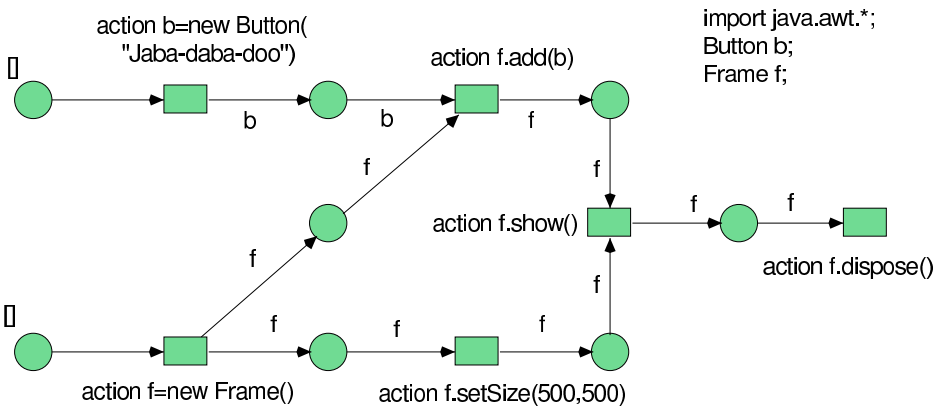


Figure 3.6: The net `frame`

Fig. 3.6 shows some example method calls. The declaration node contains an *import statement* that instructs the simulator the 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.

3.5 Tuples and Unification

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

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

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

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

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

and `z` to the third.

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.

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

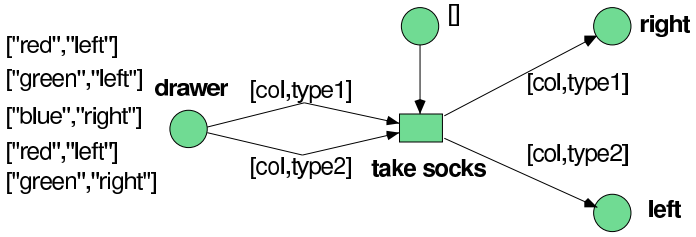


Figure 3.7: The net `socks`

In Fig. 3.7 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.

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. 3.8 and 3.9 you can see a simple example. These nets are available in the `samples/creation` directory.

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

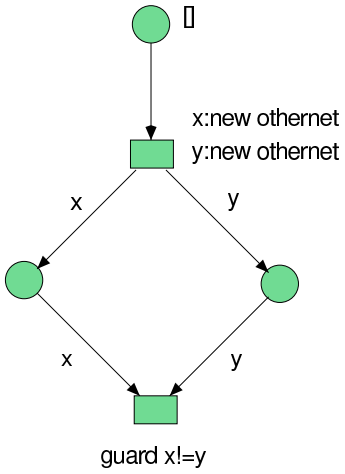


Figure 3.8: The net `creator`

Figure 3.9: The net `othernet`

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 coloured Petri nets by Christensen and Damgaard Hansen in [1]. 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`, which 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 downlinks. An example downlink 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`.

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. 3.10 you can see an example net with local channels, which is provided in the directory `samples/channel`. The input place of the left transition is marked and the transition's downlink specification can be met by synchronizing with the right transition. Both transitions fire synchronously, such that one token is removed from the left place and one token is added to the right place in a single step. Now no more transitions are enabled. The left transition lacks a token on its input place, the right transition has an uplink that is not invoked by another transition.

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

It is allowed that a transition has multiple downlinks. It is also allowed that a transition has both an uplink and downlinks. This is exemplified in Fig. 3.11. Again the transition on the left initiates the synchronization. The required

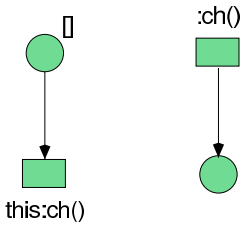


Figure 3.10: The net `synchro`

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.

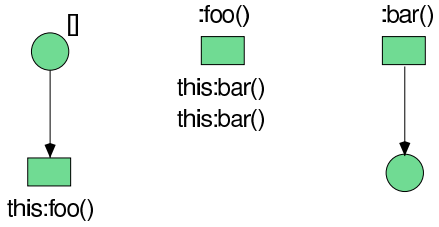


Figure 3.11: The net `multi`

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. 3.11 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 3.12: The net `param`

In the previous examples we only encountered local synchronizations within one net, but Figs. 3.13 and 3.14 show two separate nets that can communicate. The net represents the basic schedule of Santa Claus on the night before Christ-

mas. 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 childrens' boots, maybe some candy or a brand new game.

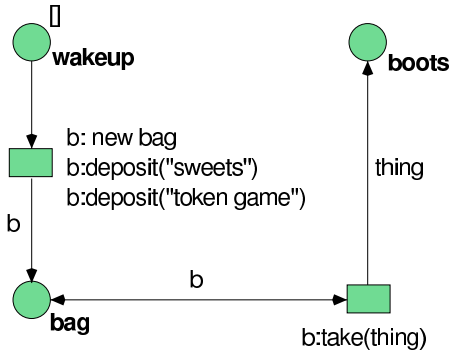


Figure 3.13: The net `santa`

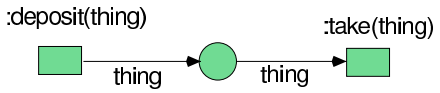


Figure 3.14: The net `bag`

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.

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 arbitrary ways.

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*.

The net from Fig. 3.15 models a very simple bank account. The customer can only deposit and withdraw money and view the current amount. But we

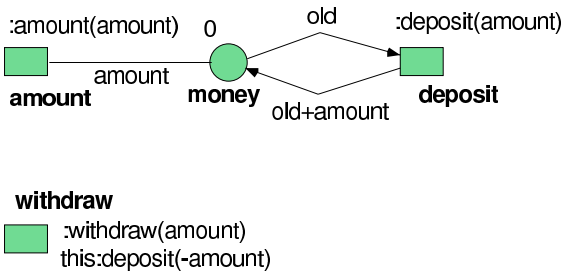


Figure 3.15: The net `account`

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

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

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

As in reference nets, variables need not be declared. If variables are declared, they must be declared before they are used. In our example there are no variables except for the input parameters and the special variable `return`, which

is used in the last method `currentAmount()`. This variable is automatically declared in each method that has a non-void return type. A non-void method returns the value of `return` at the end of its body.

The stub description can now be compiled with the command

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

from the Unix command prompt, if 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. In Fig. 3.16 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.

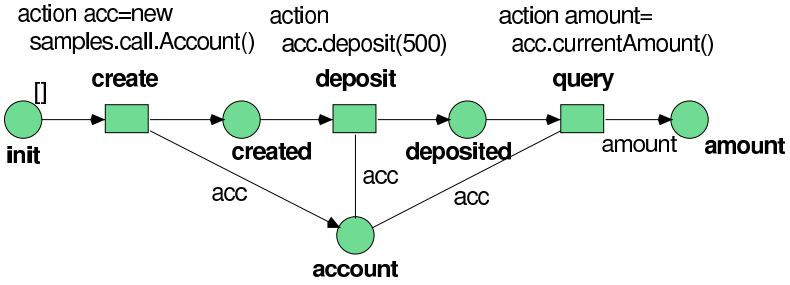


Figure 3.16: The net `customer`

If you load the two nets 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
(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 press will trigger an `java.awt.event.ActionEvent` that is processed 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 [7].) 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. 3.17 and 3.18.

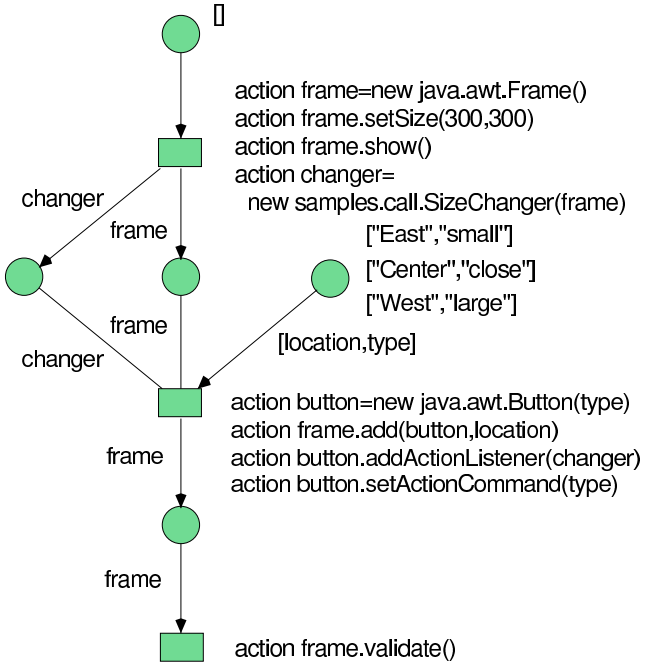


Figure 3.17: The net `buttonmaker`

The net `buttonmaker` is used to construct the frame with buttons and `SizeChanger` objects. Later on, each mouse click on one of the three buttons results in an event that is propagated to the `sizechanger` nets. 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 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.

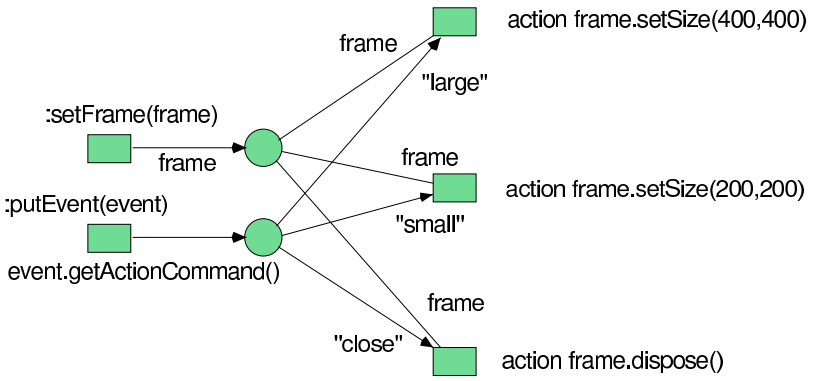


Figure 3.18: The net `sizechanger`

```
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.

```
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 all 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. We must still draw the associated net, though.

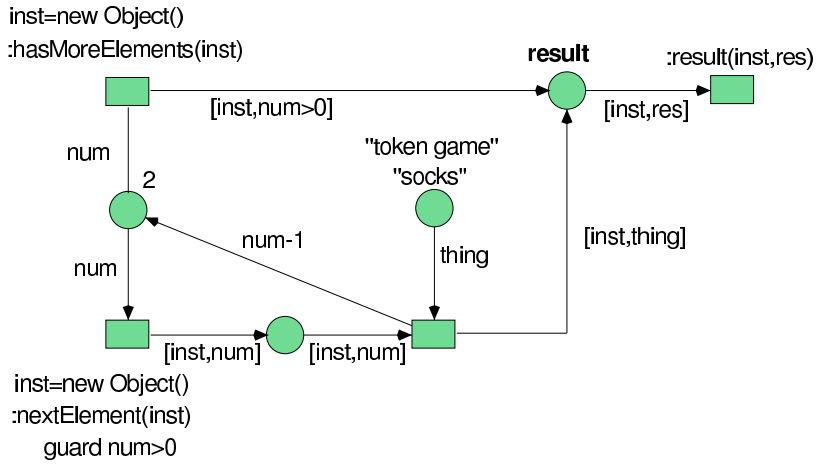


Figure 3.19: The net `enumbag`

Fig. 3.19 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. 3.20 you can see Santa distributing the Christmas presents.

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

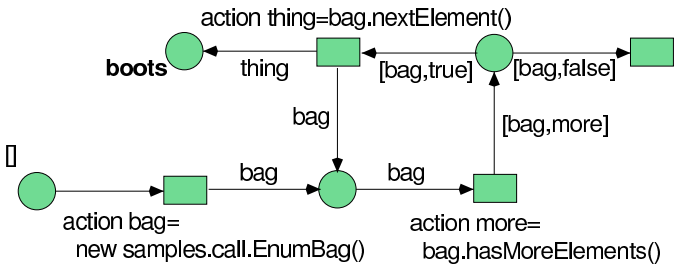


Figure 3.20: The net `enumsanta`

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 Pitfalls

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

3.9.1 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.

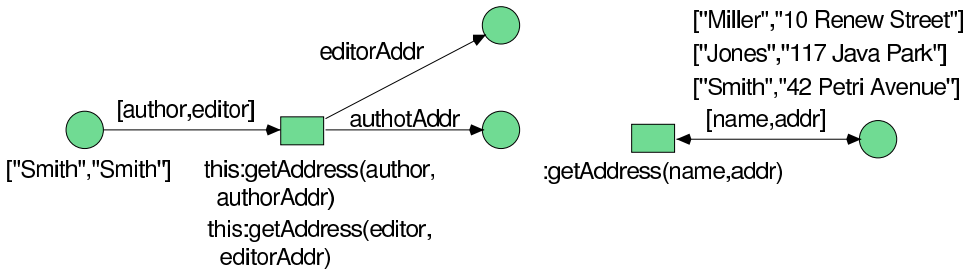


Figure 3.21: The net `reserve`

The net from Fig. 3.21, which is filed in the directory `samples/pitfalls`, shows a small excerpt of a workflow when a printing error is found in an article in a book. The desired effect is to lookup the address of both author and editor, so that they can be sent a notification. The modeler wanted less net elements, therefore both lookups were done by a single transition.

However, the database access is done with a reserve arc, so that this procedure fails when author and editor coincide. It would have been better to use a test arc in this place, because there is no need to reserve the information in the database. Such errors are especially difficult to detect when they are hidden behind a synchronous channel invocation.

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

3.9.2 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.

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

3.9.3 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.

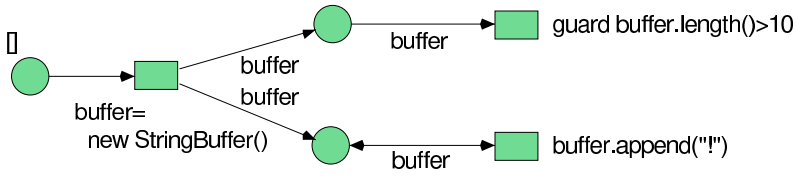


Figure 3.22: The net `buffer`

In Fig. 3.22 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 it 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.

3.9.4 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. The only way to load the new classes is to close and restart Renew entirely.

3.9.5 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

Drawing the Nets

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 [3]. 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 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.

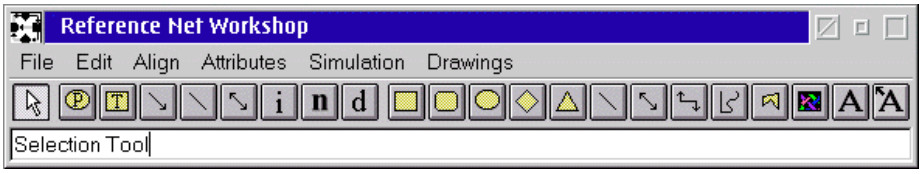


Figure 4.1: The Renew Window

There is always one active drawing window. Selecting a pull-down menu invokes a command which affects the active window, its drawing, or a selection of figures of that drawing, unless it has a global effect only. Examples of menu commands are saving or loading a document or changing attributes of figures. The menu commands are explained in Section 4.3. On the other hand, the toolbar is used for selecting a current tool. With a tool you can create or edit

certain kinds of figures in a drawing. All tools available in the toolbar are discussed in Section 4.2. Since each tool (but the selection tool) is related to a certain type of figures, the corresponding figure type is also explained in that section. To manipulate figures, handles are used. Handles are small squares or circles that appear at special points of a figure when the figure is selected. Dragging 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 2.3. You should then be able to start Renew from the command line, just typing `renew`, or using a programm 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.

4.2 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.

4.2.1 The Selection Tool

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

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

Select By clicking on a figure, it becomes selected. A selected figure can be recognized by its visible handles. Depending on the type of figure, different

handles appear, but in all cases, some handles will appear. There are even non-functional handles, which are just there to show that a figure is selected and do not have any additional (manipulation) functionality. If another figure is selected, the current figure becomes deselected. To clear the current selection, click inside the drawing, but not on any figure.

Add to Selection If the shift key is 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. Most figures have an alternate function that is invoked by clicking on them with the right mouse button. All text figures, for example, are switched into edit mode, while for other figures, their first connected text figure child (see Section 4.2.2: The Connected Text Tool) is selected if there is any.

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 4.2.2: 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 4.2.2 in the subsection of the corresponding figure's tool.

4.2.2 Drawing Tools

Renew provides several drawing tools which create and manipulate drawing figures. These drawing figures do not have any semantical 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 favourite drawing tool (like adjusting line width and such), but remember that Renew is a Petri net tool, not a drawing tool in the first place.

The Rectangle Tool

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

After a new figure has been created, the new figure is not automatically selected. To do so, just click on the figure with the selection tool (see Section 4.2.1). Now, the figure's handles appear. In the case of a rectangle or ellipse figure, these are sizing handles which are displayed as small white boxes at the corners of the figure. These handles let you change the dimension (and location) of a figure after you created it. Depending on the position of the handle, only certain changes are allowed. For example, the "east" sizing handle only allows to change the width of a figure, while maintaining the location of the left side, and the "south-west" sizing handle only lets you relocate the lower left corner of a figure, while maintaining the location of the upper and right side.

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

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

The Round Rectangle Tool

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

The Ellipse Tool

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

The 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 colour, but it respects the pen colour (see Section 4.3.4).

The Connection Tool

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

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

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

An advanced feature is to produce *intermediate points* in a connection. Activate the connection tool and click on a intermediate point of 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".

I am really sorry to tell you that in this version of Renew, you cannot get rid of intermediate points.

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 colour. 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 4.3.4). 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. After that, continue like with a normal text figure (see above).

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

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

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

parent again, its old parent (if there is any) is selected in the same manner, to indicate if you let go the mouse button now, the parent will stay the same.

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

The assignment of a parent even works for non-connected text, but be careful: There is no way to set the parent of a text figure back to "none", in other words to convert a connected text figure to a text figure.

4.2.3 Net Drawing Tools

Now it is really getting interesting: This group of tools allows you to draw Petri nets that have a 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 4.4).

Since all net element figures (transitions, places, and arcs) may have inscriptions, Renew supports automatic inscription selection and generation. Click on a net element figure with the right mouse button, and its first inscription figure is selected. If there is no inscription figure, a new one is created with the default inscription `x`. This is quite convenient, as most simple inscriptions are arc inscriptions with a single variable. Of course, you have to change the inscription for transitions and places.

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 4.2.3: The Arc Tool).

The new handle has a special behavior when you do not stop dragging on an appropriate figure. 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 an 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 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 4.2.3: The Transition Tool).

There are three arc tools for different arc types:

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.

TestArc 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 3.9.1). This means it does not matter whether you start a test arc at the place or at the transition.

ReserveArc 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 3.9.1.

Using the `AttributesArrow` menu, it is possible to change the type of an arc after its creation.

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.

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

Again, in contrast to text figures, inscription figures have a 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 4.4). Refer to Section 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 that the simulation trace becomes more readable. 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 (see Section 4.3.5).

Each place and transitions should have at most one name figure connected (but the editor does not check this). 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 contents of a declaration figure is syntax-checked as soon as you stop editing. For an explanation of syntax error that may occur, refer to Section 4.4.

4.3 Menu commands

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

4.3.1 File

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

New 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 4.3.6).

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 4.3.6) 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 4.1) to a file using a textual format. The drawing is saved to the last file name used, which is the file it was loaded from or the file it was last saved to. If the drawing has not been saved before, this function behaves like `Save Drawing as...` (see below).

If there is an old version of the file, it is overwritten. Depending on your operating system, overwriting a file might need reconfirmation 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 in future invocations of `Save Drawings`. 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 4.3.6). 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 4.3.6). Be careful: Unlike most other applications, Renew does not require a confirmation to close an unsaved drawing! So save first, then close!

If you close the last and only drawing in Renew, there is a known bug that results in error messages in the console window when you select tools and invoke certain menus, until you open a new drawing window.

Print...

The print menu invokes an platform dependent print dialog and lets you make hardcopies of the active drawing. Using the Java standard print systems, though, the quality of the printer output is usually very poor. This is why we implemented a Java PostScript output feature (see next section).

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

Export PostScript...

This function produces PostScript code that resembles the current drawing. You have to select the file 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). As a PostScript description of one DIN A4 page is generated, you might want to use a tool like `ps2epsi` to convert the resulting file to the Encapsulated PostScript (EPSI, sometimes just called EPS) format. This format 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 `CH.ifa.draw.util.PostScriptWriter` (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.

Export to Macao...

Exports the active drawing in the Macao format. Macao (see [6]) 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 [2]) 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-coloured Petri nets without synchronisations, 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.

Exit

Exits Renew immediately. Like in Close Window, there is no confirmation, so be sure you have saved everything you need later on!

4.3.2 Edit

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

Cut, Copy, Paste

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

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

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

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

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

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 connction 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 4.2.2: 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 (that are all selected).

Send to Back, Bring to Front

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

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

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

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.

4.3.3 Align

The Align menu allows to align a figure's position according to a grid or to other figures.

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 customisable.

Lefts, Centers, Rights

These functions all align the selected figures' 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.

4.3.4 Attributes

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

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 4.2.3: The Inscription Tool). A `Name` text type does not change the simulation, but makes the log more readable (see Section 4.2.3: The Name Tool).

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

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

Trace

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.

4.3.5 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 4.3.6). The drawing window containing the net that is to receive the initial `new()` synchronisation has to be activated.

Some syntax checking is done even while you edit the net (see Section 4.2.3: The Inscription Tool). When you try to run a simulation of your reference nets (see below on how to do this), first the reference net compiler is invoked and may report further errors (see Section 4.4). You have to correct all compiler errors before you can start a simulation run.

During simulation, there is textual and graphical feed-back. The Java console prints a log of the simulation that is briefly described in Section 4.3.5. In this log, you can see exactly which transitions fired and which tokens were consumed and produced.

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]`.

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. This current marking window shows the name of the 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, scroll bars will appear. A current marking window contains an entry for all different token values and tells you the multiplicity for each. The token value is 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.

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!

Run Simulation

This function starts or continues a simulation run that continues automatically until you terminate 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.

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 already a simulation is 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`.

Terminate Simulation

This menu 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.

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

4.3.6 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.

Unfortunately, a window that was "iconised" (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 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 4.2.2: The Text Tool.

Renew displays exactly one error at a time. If a second error is found, the old error message will be discarded and the new error message will be displayed in the error window.

Some errors are not reported at the place where they originate. E.g. if you are using a declaration figure, an undefined variable is detected when 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.

4.4.1 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.

4.4.2 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?

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.

4.4.3 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 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.

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?

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?

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?

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 partly 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 0.8 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, but not free and with a slightly uncertain future) to ANTLR (a great program, but free and available with source).

Bibliography

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

[2] Eindhoven University of Technology. *Woflan – The Workflow Analyser*, 1998.

WWW page at `http://pceric.win.tue.nl:1610/woflan/`.

[3] Erich Gamma. *JHotDraw*, 1998.

HTTP file at `http://members.pingnet.ch/gamma/JHD-5.1.zip`.

[4] 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.

[5] 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/`.

[6] lip6. *Macao Graph Editor Framework*, 1998.

WWW page at `http://www-src.lip6.fr/logiciels/macao/`.

[7] Peter van der Linden. *Just Java*. Prentice Hall, 1996.

Appendix A

Contacting the Team

To get in contact with us, you can send an email to

`support@renew.de`

regarding any aspect of the Renew tool, especially update notification requests, bug reports, feature requests, and source code submissions. Our postal address is

Arbeitsbereich 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 Dough 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. This grammar was contributed to JavaCC by Bill McKeeman (`mckeeman@mathworks.com`). 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 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.2 Original Parts

This copyright section deals with those part of Renew that are not based on other works, i.e. the packages `fs` and `de.renew` without the JavaCC grammars, some sources in `CH.ifa.draw.cpn`, and the example nets.

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 and net drawings) 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 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.