# A Framework for Interacting Design/CPN- and Java-Processes

Olaf Kummer Daniel Moldt Frank Wienberg

Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg
{kummer,moldt,wienberg}@informatik.uni-hamburg.de

**Abstract**

In order to widen the applicability of Design/CPN for the specification and design of large scale distributed applications, a framework has been developed that supports the interaction of Design/CPN and Java processes. Thereby a seamless embedding of the two worlds of Petri nets and object-oriented programming is achieved, allowing problem oriented modeling at different abstraction levels in a fully distributed environment.

The general possibilities to connect Design/CPN with remote processes are discussed and a specific implementation of the required framework is sketched. Promising application areas are named and for some of them concrete example models are provided.

**Keywords:** Coloured Petri Nets, Design/CPN, Distributed Simulation, Framework, Java, Workflow, Prototyping

## 1 Introduction

The specification of systems, its evaluation, and its transfer to implementation is still a major task for computer science. One very promising technique in the area of specification, especially when concurrent and distributed systems are involved, are Coloured Petri Nets (see [7]). Because a strong interconnection of specification and implementation is very useful when developing a system, it is desirable to bring together the worlds of Coloured Petri Nets and some popular programming language.

In the area of implementation Java (see [6]) aroused special interest for building applications for the Internet, as it is an object-oriented, reasonably portable programming language that supports additional features such as high-level networking and easy multi-thread programming.

As a specification tool based on Petri nets, we chose Design/CPN (see [3] and [9]), because it is flexible and powerful and comes with a specially adapted graphical editor. Design/CPN supports the development of large systems by means of hierarchical models. Furthermore, an internal programming language, namely ML, can be used to extend the tool for our needs.

In the environment proposed here, Java should be used for the implementation of graphical user interfaces, database connectivity, and other applications for which one can fall back upon reusable implementations, while Design/CPN serves as the graphical specification tool that is powerful in designing and executing models of concurrent, distributed systems.

But there are more reasons why an interaction of Design/CPN and Java has been expected and indeed turned out to be fruitful, namely to overcome some limitations of Design/CPN: The tool is designed for single-user mode only, but especially large-scale projects are dependent on group- or teamwork. Although there is a flexible hierarchy concept, Design/CPN does not really support component re-use, as the exchange of parts between different models is difficult. In the simulator, interfaces for calling programs implemented in languages other than ML are supported on a very low level only. Although the tool itself, especially with the extension Mimic (see [13]), offers graphical user interface routines, these are also very rudimentary compared to state-of-the-art tools. Last but not least the process of implementing a system for which a Design/CPN prototype has been designed is not well supported by the tool. Since there is no way to execute a Design/CPN model in a stand-alone fashion and also simulation of large-scale models is not quite as efficient as a (compiled) program, it would at least be desirable to provide a stepwise migration of the system from net models to some programming language.

Motivated by all these considerations, a framework has been designed that establishes new possible fields of application where Design/CPN may be employed. The framework extends Design/CPN in several ways: Distributed simulation is achieved by different technical means, namely sockets and pipes. The implemented architecture and alternative concepts for the interconnection of multiple Design/CPN processes are described in section 2. In section 3 it is shown how Design/CPN can communicate with Java processes during simulation. This allows invoking Java methods and even the creation of Java objects. Section 4 explains the reverse communication direction where Design/CPN processes are invoked by Java, viewing the whole Design/CPN process as an object. Possible applications of the extensions and benefits gained from them are discussed in section 5. Section 6 provides an outlook on how the framework can be extended further and how it can be exploited for other approaches.

## 2 Distributed Design/CPN Processes

If we want to achieve a distributed simulation using Design/CPN, we have to run multiple instances of the program and allow synchronization of the multiple nets that are being executed.

Distributed simulation in our sense does not mean the distributed simulation of a single Petri net, which would imply the distributed solving of conflicts. This problem has been treated in e.g. [5]. Instead, we will distribute our models in such a way that only local conflicts can occur, so that they can be handled within one Design/CPN process.

### 2.1 Possible Communication Channels

Running multiple instances of Design/CPN in a single simulation run requires communication and synchronization. Since shared memory is not provided, we have to use some form of message passing.

Whatever I/O-channel is used, a blocking I/O call will stop the current net simulation, because the simulator is strictly single threaded. Hence it is usually not advisable to perform blocking I/O, although this might be an option in some cases. Instead, we have to do polling I/O, checking for new input from time to time.

There are essentially three possibilities to send and receive messages from within a Design/CPN simulation process:

- Ordinary files. One Design/CPN process might write to a file, while another process might read the file. This is an inefficient method, especially if the processes access a file by NFS. Almost all network file systems cache file contents so that a change does not immediately propagate through the whole network. This makes ordinary files practically worthless for communication purposes.

- TCP connections. A TCP connection involves a server and a client. The server will setup a server socket under a well-known port number, so that it can accept an arbitrary number of connections from clients. Each server socket is identified by the port number and the server's host at the time the client socket is created.

  Using the ML library `SysIO` Design/CPN supports client sockets with the help of low-level file descriptor I/O.

- Pipes. In Design/CPN it is possible to generate an external process on the same machine by means of the `execute` function. The call will provide input and output pipes from Design/CPN to the created process. These can be handled by the usual ML stream I/O library, which is significantly easier to use than the raw file I/O. Of course, after a few routines have been written to handle the interprocess communication, the designer of a model does not get into contact with these routines very often anyway.

  This approach is suggested in the `process` example that is distributed with Design/CPN 3.02 which starts a Tcl/Tk process as an example.

One the one hand, pipes are most useful when we want to communicate with other processes on the local machine that can be started during the simulation. On the other hand, the more versatile TCP connections have to be used if the processes is required to run on different machines or to be started before the net simulator.

### 2.2 Possible Communication Architectures

The process started by an `execute` could of course be another Design/CPN process, but a Design/CPN simulation cannot access its own standard input and output streams. Moreover, at present there is no way to start a simulation automatically within a Design/CPN process with which we could interact. These two difficulties practically rule out direct pipe communication between the two nets.

Moreover, we cannot use direct TCP communication, because it is not possible to implement server sockets within Design/CPN. This leaves us with three basic options:

- One dedicated communication process is started (the communication server). It opens a server socket that can be accessed by an arbitrary number of Design/CPN processes with one TCP connection each. For a visualization of this architecture see Fig. 1.
- Every Design/CPN process starts a messenger subprocess and accesses it via pipes. The messenger subprocesses is responsible for transporting the messages to and from the server. It will handle the protocol with the communication server as before (see Fig. 2).
- Only the subprocesses are present and implement a suitable algorithm for direct message exchange, typically using TCP (see Fig. 3).

In principle it would be possible to organize the processes like in Figs. 2 or 3 using TCP connections instead of pipes. However, this would imply programming more difficult ML functions without any obvious benefits.

Of course the communication processes can be implemented in any language that supports the necessary networking ability, but among the many options the most promising seems to be Java for the reasons given in section 1.

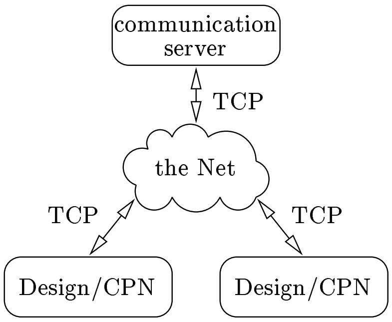


Figure 1: Server solution

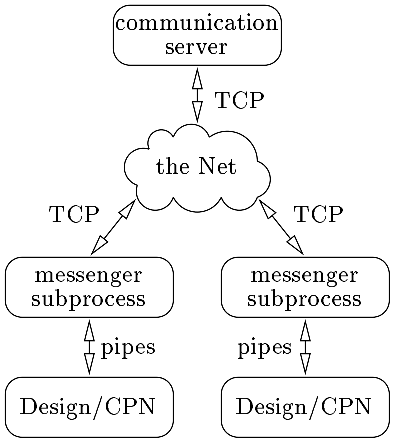


Figure 2: Messenger solution

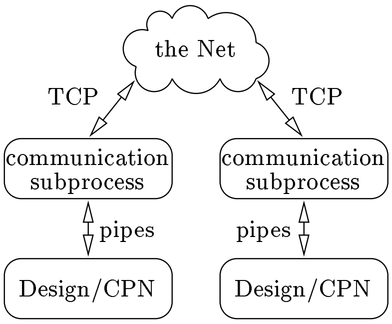


Figure 3: Fully distributed solution

### The Server Solution

In Fig. 1 we can imagine the Design/CPN processes as actors that exchange messages via a mailbox (the server). Each actor has a unique mail address, which may be generated by the system at runtime and which will usually exploit the unique network name of the local computer. Some actors may also be assigned a unique well-known address chosen at programming time.

In its simplest form, the mailbox only has to provide commands to send a message to a mail address and to receive a message, if one is available. For performance reasons it may be useful to receive all pending messages. More complex implementations would allow a blocking wait for messages, but this is of little use for a Design/CPN process, since it would suspend the simulation completely. Other options include a forwarding mechanism or a test if a mail address is valid.

The host and the port where the mailbox is located must be public for all Design/CPN processes, so that they can establish a connection.

The mailbox can quickly become the system's bottleneck, especially because polling access is required on the side of Design/CPN. Nevertheless, this method is surprisingly practical and will perform well provided there are only few processes.

### The Messenger Solution

In this case there is still a mailbox, but every Design/CPN process starts a special messenger process by means of the `execute` function, as in Fig. 2. The messenger process can communicate with Design/CPN locally, which is usually much faster than remote interaction. Also, since the messenger is only responsible for a single Design/CPN process, we can use a significantly simpler protocol. Moreover, the messenger can communicate with the mailbox using efficient blocking I/O, thereby taking some load off the bottleneck. Of course, the price to pay is a further indirection leading to some communication overhead.

### The Fully Distributed Solution

Here we no longer have a central instance; instead each subprocess spawned by Design/CPN interacts directly with other subprocesses. This would lead to further performance benefits at the price of a vastly increased complexity of the message-handling algorithm.

Even in a fully distributed architecture there might be a kind of centralized name service, which would allow the distributed nets to communicate without knowing the actual location of the other nets.

## 2.3 A Communication Package

We have tried the server solution (in C) and the messenger solution (in Java). Here we are going to limit ourselves to a description of the messenger solution, because it results in a more readable Design/CPN package.

The actual implementation of the mailbox and the messenger processes are beyond the scope of this paper. Let it suffice to say that much effort went into the handling of concurrency and into the protocol. The actual network programming turned out to be very easy in Java, contrary to C where network calls consitute the majority of the code.

At the moment the pure subprocess solution is not implemented, but it could be done in about three weeks, if this is required due to performance reasons. The Design/CPN library, which we are going to describe now, and the protocol on the pipes would not change at all, making an upgrade possible without any noticeable difference for the users.

All basic message handling is done on a single I/O page that can be reused in every distributed system of nets, see Fig. 4. It encapsulates all the transitions that set up the connection and send or receive messages.

In the communication package described here every Design/CPN process is assigned an arbitrary character string that can be used as the mailbox address. The I/O page is given the mailbox address on which it will listen via the "in" port place `connect`.

The ML variable `messenger` has to contain the path and name of the executable that starts a messenger process and should be defined in the global declaration node, e.g.

```
val messenger = "/home/cpnuser/bin/messenger";
```

This path name is not passed as a token, because it does not often change and because an incorrect path might result in errors that are undetectable in the ML code.

Outgoing messages must be put into the "in" port place `send` in the form of an address/data pair. Both will be handed on to the messenger through the output stream, preceded by the command `send`. Messages are received from the messenger through the input stream. It is important to test whether there is any pending input with the ML function `can_input`, because `input_line`[1] suspends until a newline character (`\n`) is read. The page provides access to the received messages using the "out" port place `received`. The data to be sent and received is always a character string. Other datatypes must be converted to a string format, e.g. by using the predefined `mkst_col'`*colour* functions.

Tokens can be put into the `polling` place or be removed from there in order to either start or stop polling for new messages. Design/CPN only allows to stop an "automatic"

[1]The function `inputLine` used here is just a customized version of `input_line` that cuts off the newline at the end of the result string.

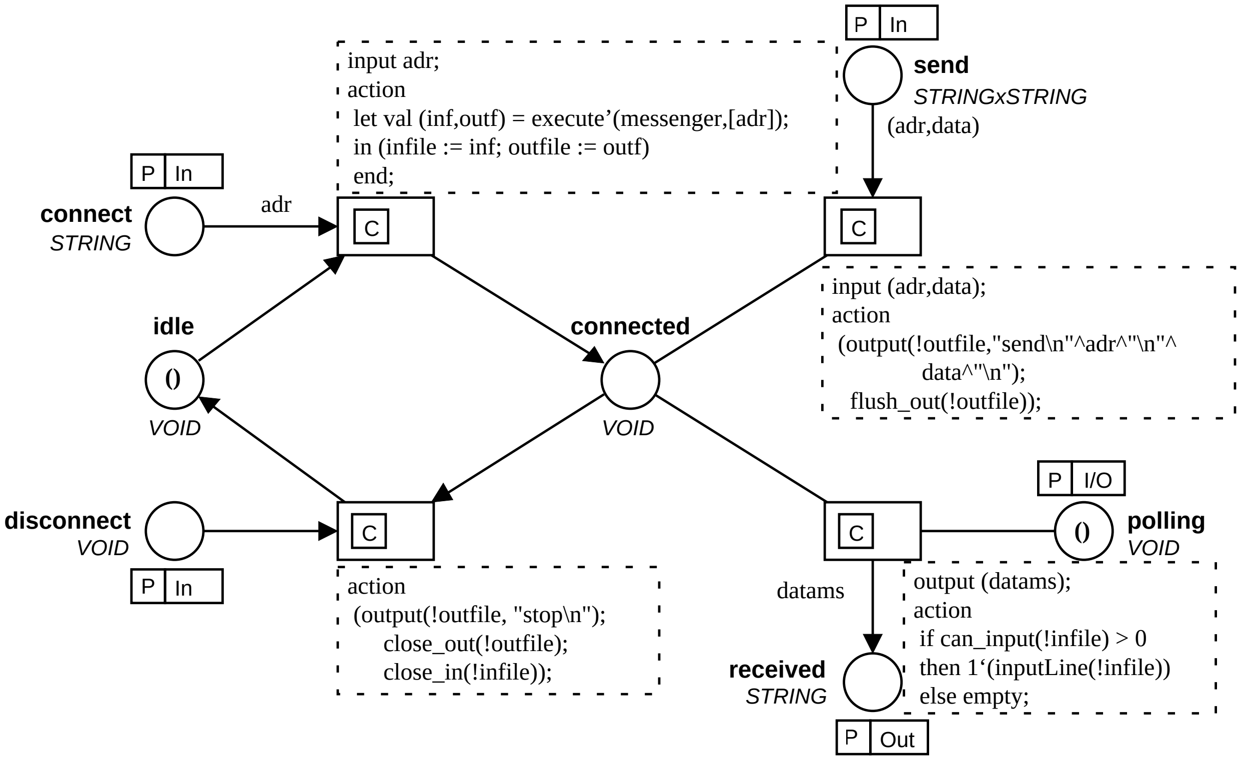


Figure 4: A subpage for string message I/O (`IOPage`)

simulation run after a given number of steps or in case there are no more enabled transitions. Removing the `polling` token offers the possibility to stop an automatic simulation run without disconnecting. Furthermore, this feature should be used with nets where it is known when to expect incoming messages, e.g. only as answers to query messages that have been sent before. It is much more efficient not to poll for new messages while the net is working locally and not expecting any message input. If the `polling` port place is not assigned on the "parent" page, the initial marking of one token will be used and polling will always be activated as long as the connection is up. It should be recalled that Design/CPN uses the initial marking of a port place only if no socket place is assigned to it.

Finally, there is a facility to close the connection and shut down the messenger process, because these processes might stay alive when they are not terminated properly. If one wants to do so, one has to change the marking of the `disconnect` place to one token and fire the `disconnect` transition.

Because pipes cannot be uniquely represented as strings, they cannot be used as token colours, hence they have to be stored in reference variables. These are not allowed to be used in arc inscriptions, but in code regions only. We defined two *global* instance variables, `infile` and `outfile`. If one wants to use the subnet more than once, because there is some need to run multiple messenger processes, one can also use *instance* reference variables which have different values for each page instance. This may be desired when many net fragments are developed and tested within one simulator, before they are finally split into many independent nets. The reason why we used global reference variables is that we normally do not use more than one instance of the I/O page and also that instance variables cannot be reached from ML code via Design/CPN's *ML evaluate* feature, e.g. to close the streams manually.

# 3 Accessing Java from Design/CPN

It should be clear that the message passing scheme used in our architecture does not rely on specific Petri net techniques, so that Design/CPN could be complemented by programs in arbitrary languages that support TCP. Again we choose Java as our example language, even though experiments have also been done with C (for details see [1]).

A straightforward implementation would provide only the basic routines to send and to receive string messages, leaving the programmer with the task of making the necessary calls. But we can increase the developers' productivity by defining a standard message format, so that we can supply reusable parsing algorithms.

On top of that, we can provide algorithms that actually perform the call that was requested by Design/CPN, so that the message passing framework becomes completely invisible to Java programs. Quite the same can be achieved on the side of Design/CPN where we make a Java method call look like an ordinary substitution transition (see section 3.4).

## 3.1 The Message Format

The message format should contain all the necessary information to make a Java method call: the object whose method is invoked, the caller that awaits a return message, the method name for the called Java object, and a list of parameters. We have to distinguish these *call messages* from another type of messages which we call *return messages*. A return message is much simpler, as the method name and the caller can be omitted. What remains is just the target object (which is the sender of the call message) and a list of parameters, which in case of Java may only be of length one or zero (if a method is of return type `void`).

One suggestion for a suitable message format that has been implemented has the main aim to keep the net inscriptions and functions simple on the side of Design/CPN. As Design/CPN is based on the functional language ML, it can handle lists very well. Thus, a message is implemented as a list of the components mentioned above.

However, Design/CPN-colours are strongly typed, so a union type of all different types that may appear in a message has to be defined. We end up with a message colour definition as follows:

```
color OBJECT = union Null
                   + RC:RCLASS
                   + RI:RINSTANCE
                   + M:METHOD
                   + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL;
color MSG = list OBJECT;
```

with `RC` being a colour that represents a reference to a remote class, `RI` representing a reference to a remote instance, `M` declaring a method name and `Int`, `Str`, `Bool` and `Real` being the constructors for the basic datatypes available in ML and Java. Other types, especially arrays, could be added, if desired.

This very general message format needs additional constraints for well-formed call and return messages, but offers the advantage to define both with the same Design/CPN colour. In order to distinguish call and return messages easily, we chose to put the method name in the first position of the object list instead of using the order of the object-oriented dot notation, where the receiving object is named first. A message starting with a method name is assumed to be a call message, or else a return message. The complete sequence of a call message is

```
[M(method-name), invoked-object, caller-object, param1, param2, ... ]
```

A return message is simply

```
[receiver-object, result]
```

where *result* is optional and the *receiver-object* is the former *caller-object*.

References are defined as tuples of a class and a process identifier the latter of which is used to locate remote objects. Unlike a class reference, an instance reference also contains an ID as a third component that makes the triple a globally unique identifier.

```
color RCLASS = product CLASS * PID;
color RINSTANCE = product CLASS * PID * RID;
```

These tuples could have been coded into a single string, but using tuples we can apply the built-in Design/CPN pattern matching capabilities to select the information that is needed to send and receive messages.

All colours that have not been declared are simply defined as `STRING` in our implementation.

Alternatively, the structure of a message given at the beginning of this section could be directly translated into Design/CPN-colours. For a call message, this would result in a four-tuple of the object being invoked, the method name, the calling object and a list of parameters, which again is a list of objects defined as the message above.

### 3.2 A Sample Message Communication

To illustrate the message format, we will provide a code fragment using the big integer numbers provided by the standard Java library. Real-world applications will of course access more complex classes, but the example is sufficient for illustration purposes.

```
BigInteger bigNum1 = new java.math.BigInteger("42");
BigInteger bigNum2 = bigNum1.pow(42);
String bigStr = bigNum2.toString();
```

Obviously, one call and one return message have to be sent for each statement. When sending messages, we have to provide a unique name for the sender. We can for example use the constant class name `cpn`, construct the "mailbox" name as *hostname*:*process-ID* (of the Design/ML process), and use a sequence number for every sender being active concurrently. Thus, we end up with the first sender called `RI(("cpn","cpnhost:12345","1"))`[2].

At first, the net has to invoke a constructor method of the class `BigInteger`. To do so, we also have to provide a mailbox address where a special Java process is waiting for messages (e.g. `javaserver`) and send a `new` message like this:

```
[M("new"), RC(("java.math.BigInteger","javaserver")),
           RI(("cpn","cpnhost:12345","1")),
           Str("42")]
```

[2]The double brackets appear due to the standard Design/CPN `mkst_col` function which produces a pair of brackets for both the tuple *and* the union constructor `RC`. Although most brackets may be omitted in net inscriptions, interactive input, and any other expressions that are evaluated by ML, only this syntax is accepted by the predefined Design/CPN function `input_col` used for reading colour values from the input stream.

The message becomes much easier to read if we replace the objects by CPN variables, which normally is the case in net inscriptions:

```
[M("new"), bigIntegerClass, cpn1, Str("42")]
```

The Java program then performs some actions (which are explained in section 3.6) and answer with a reference to the newly created object:

```
[cpn1, RI(("java.math.BigInteger","javaserver","xyz123"))]
```

Let us call the new object `bigNum1`. Note that the given reference contains its class, the host (or rather: mailbox name) where it can be reached and some automatically generated ID. Now, we want to compose the message corresponding to the second statement. The net has to invoke the `pow` method of the given object with the parameter value `42` (this time as an integer). We can reuse `cpn1` as the sender, because no other return messages are expected and there is no concurrency involved.

```
[M("pow"), bigNum1, cpn1, Int(42)]
```

Java will answer with a message like

```
[cpn1, RI(("java.math.BigInteger","javaserver","abc456"))]
```

Let us call the given instance `bigNum2`. Now we convert this number to a string:

```
[M("toString"), bigNum2, cpn1]
```

Note that this method call does not take any parameters. Java responds

```
[cpn1, Str("150130937545296572356771...797056873877235893533016064")]
```

Have a look at the first message again, but this time using the alternative message format mentioned in section 3.1:

```
(bigIntegerClass, "new", cpn1, [Str("42")])
```

The only gain is that we can omit the union constructor `M` for the method name, but on the other hand, additional brackets appear. Also, we have to define a different colour for return messages. When receiving a message, it normally cannot be anticipated whether it is a call or a return message, so we would have to define another union colour consisting of call *or* return messages, which again makes the message format somewhat longwinded.

All in all, the perfect message format is partly a matter of taste, but we believe that the list format offers most advantages for Design/CPN. If you want customized net inscriptions, there is also the possibility of defining special colour sets in ML. Using your own string representation, you are no longer dependent on ML syntax. However, this approach takes more time to implement and you still have to tackle the problem of integrating CPN variables into net inscriptions.

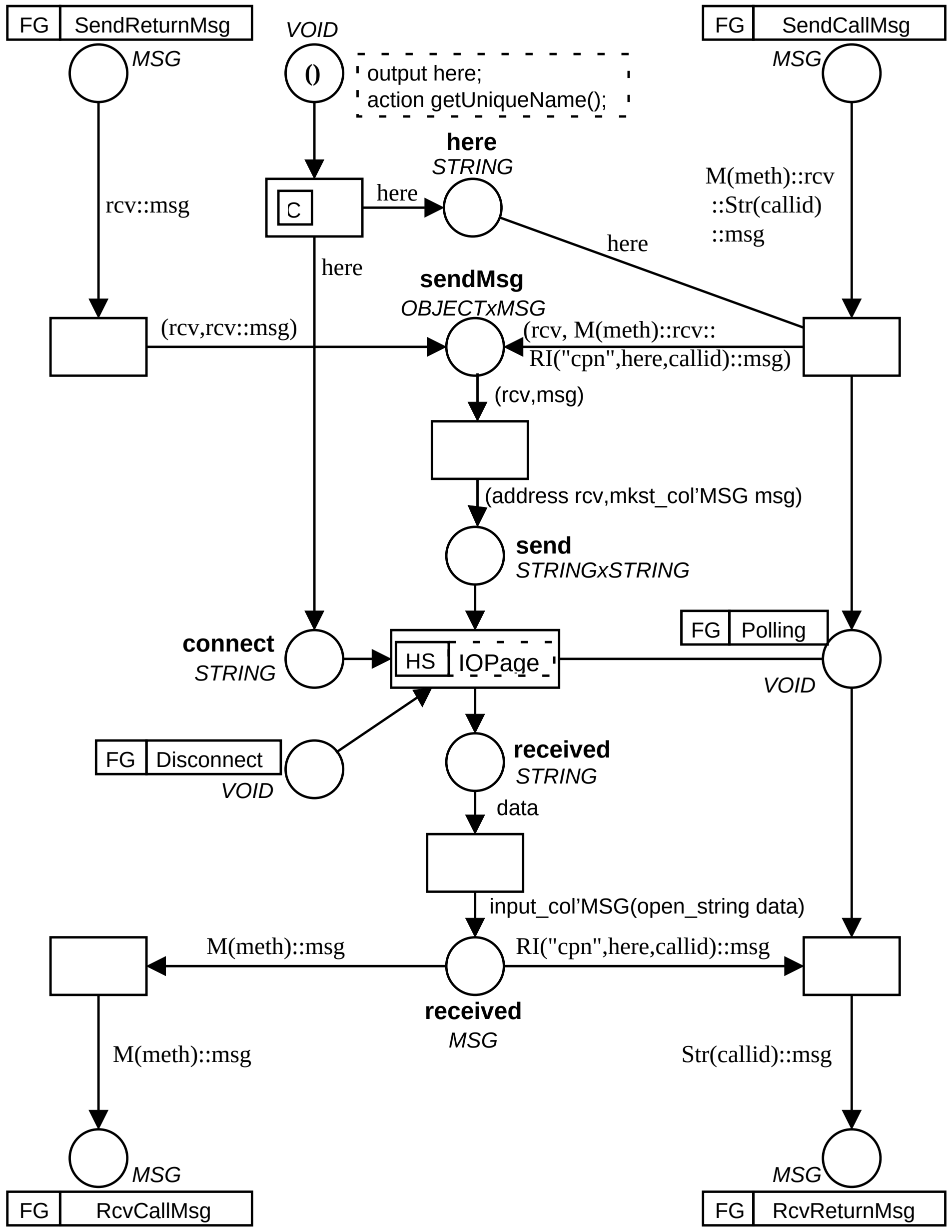


Figure 5: A message translation page

### 3.3 A Message Translation Page

On top of the message I/O page, another page is built that translates the message to and from the string representation which is used externally (see figure 5). Pages like this one must be created individually according to the input and output types that result from the conversion. The pages are expected to share a common structure, so they can be copied from a standard template and then be modified. Since most of the conversion is done by the predefined ML functions `mkst_col'`*colour* and `input_col'`*colour*, the effort is usually neglegible.

In our solution, a unique name for connecting to the mailbox is generated by a custom ML function `getUniqueName()` according to the convention described in the previous section. The `disconnect` socket is transformed to a global fusion place, so that the network

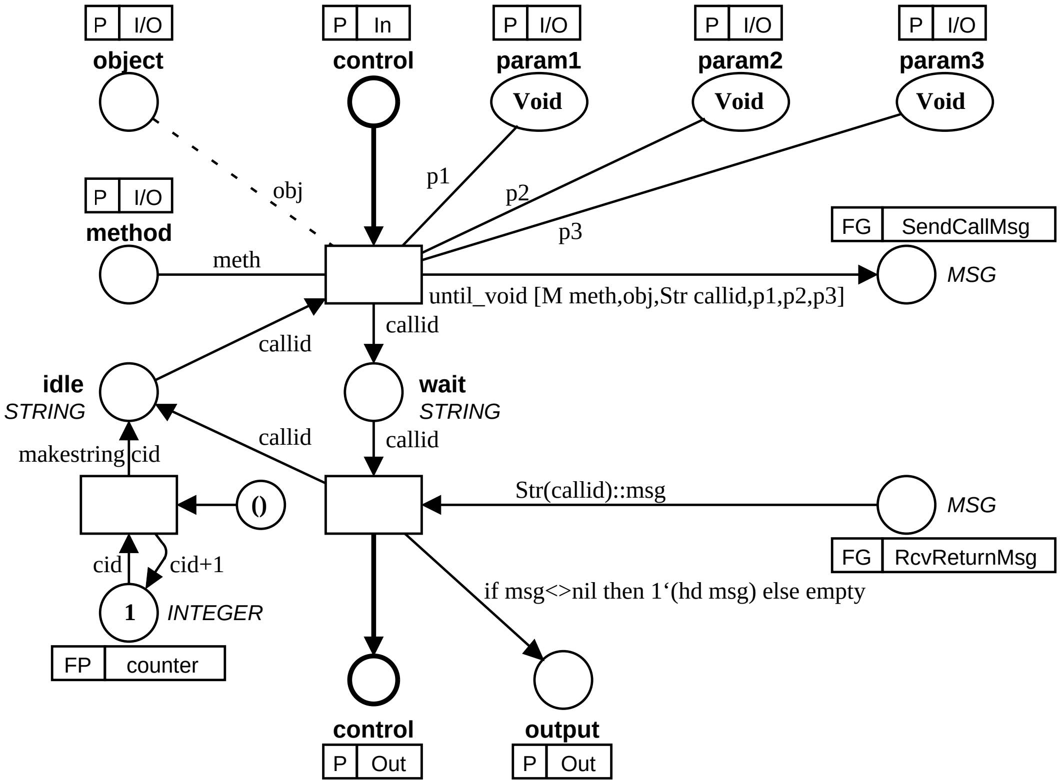


Figure 6: A subpage wrapping calls to Java methods (`Call`)

connection may be terminated from any page in the diagram. Access to the `send` and the `receive` place are provided via global fusion sets, too. Alternatively, the translation page could have defined port places, so that it could have been used as a subpage to a net that wants to send messages. We chose the former solution because the translation page is expected to be used by several other pages or page instances and it is neither necessary nor desirable to get multiple instances of this page. A single, global page is of course more efficient in simulation runs.

### 3.4 A Graphical Petri Net Notation for Method Calls

A further possibility to avoid complex arc inscriptions and to abstract from the message format is to choose a notation that is more adequate for Petri nets. A (Java) method call always goes through the same steps: The message is constructed from the components mentioned in section 3.1, with a unique identifier being constructed as the sender. Then, the message is sent and the caller waits for a return message, which is again decomposed into its components.

Figure 6 shows a subpage that implements this behaviour. The interface consists of all components of a call and the corresponding return message. The maximum number of parameters has been chosen to be three in this net but may easily be extended. No practical problems are to be expected, since the maximum of parameters needed can be determined at compile-time and should not be too high, anyway. Moreover, note the distinction between data and control flow: A special place named `control` indicates the control state of the call, while all other input data is read by test arcs[3] only. A unique ID is assigned to

[3]In fact, Design/CPN does not support test arcs that do not move any tokens. An arc with no arrows is

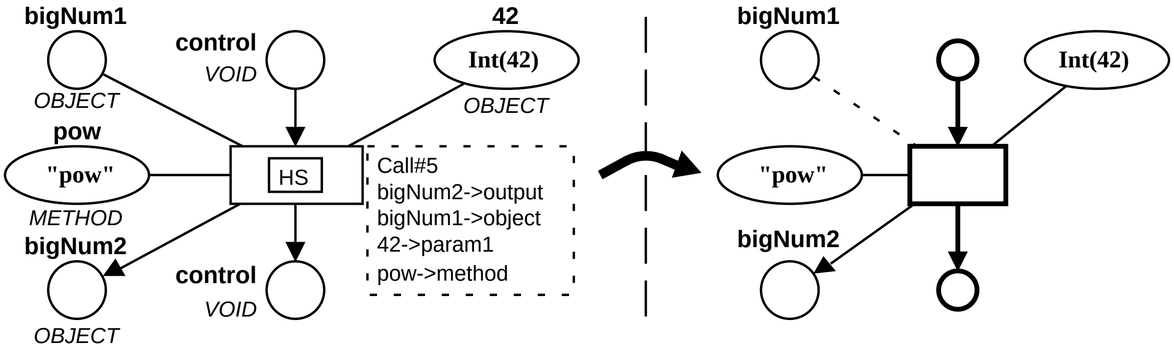


Figure 7: An example of using the `Call` subpage, illustrating the special graphical notation

each instance of the page by using the page instance fusion place `counter`. Each page instance may only invoke one method at a given time (asserted by the place `idle`), but several instances of the `Call` subpage may act concurrently. To handle parameter input, we again take advantage of the Design/CPN feature that the initial marking of a port place is only used if the port place remains unassigned. Thus, all parameter places that are not used by a substitution transition remain `Void` and are suppressed in the construction of the call message by the ML function `until_void` defined in the global declaration node. The `output` port place should only be used if the called method actually returns a result, because then the arc expression will produce no token (not even a black one). A separate control token is produced to indicate that the method call is finished. It may be used to enable the next transition.

To make the nets calling Java more readable, some Design/CPN regions have been suppressed in the following figures. However, a special graphical notation makes the syntax of the nets clear without ambiguity, as may be seen in figure 7, where the second line of Java code from section 3.2 is shown as a net. The left-hand side contains all regions and inscriptions, while the right-hand side shows how these are translated into graphical representations: All control places and control arcs are shown in bold style. They are always of colour `VOID` ("black tokens") as they store control information only. Data places and arcs are shown in normal style and all have the colour `OBJECT` unless they are a `method` place, which can be recognized by their position directly to the left or to the right to a `Call` substitution transition. If an arc has no arc inscription, it is either connected to a substitution transition (Design/CPN neglects inscription of such arcs, anyway) or it has the hidden arc inscription `()` which is one black token.

Any transition that invokes an object method, furtherto referred to as an invocation transition, is indicated by the presence of a dotted arc with no arrows. Since all these transitions are substitution transition refined by the `Call` subpage, the hierarchy substitution region (`HS`) may be omitted. In order to still state a precise port assignment, the following rules apply: The arc connecting a socket place to the `object` I/O port is exactly the dotted one. The input and output control sockets can be recognized as the bold places with input and output arcs, respectively. The input parameters are connected with test arcs (no arrows) in normal style. If more than one input parameter is used, the arcs have to be labelled with the parameter index p$n$ (this notation is not needed in this paper). The output port is assigned to the socket that is connected to the substitution transition via an output arc in normal style.

treated as an arc with arrows in both directions, thus removing and putting back identical tokens. However, this is not semantically different as long as an interleaving semantics is assumed.

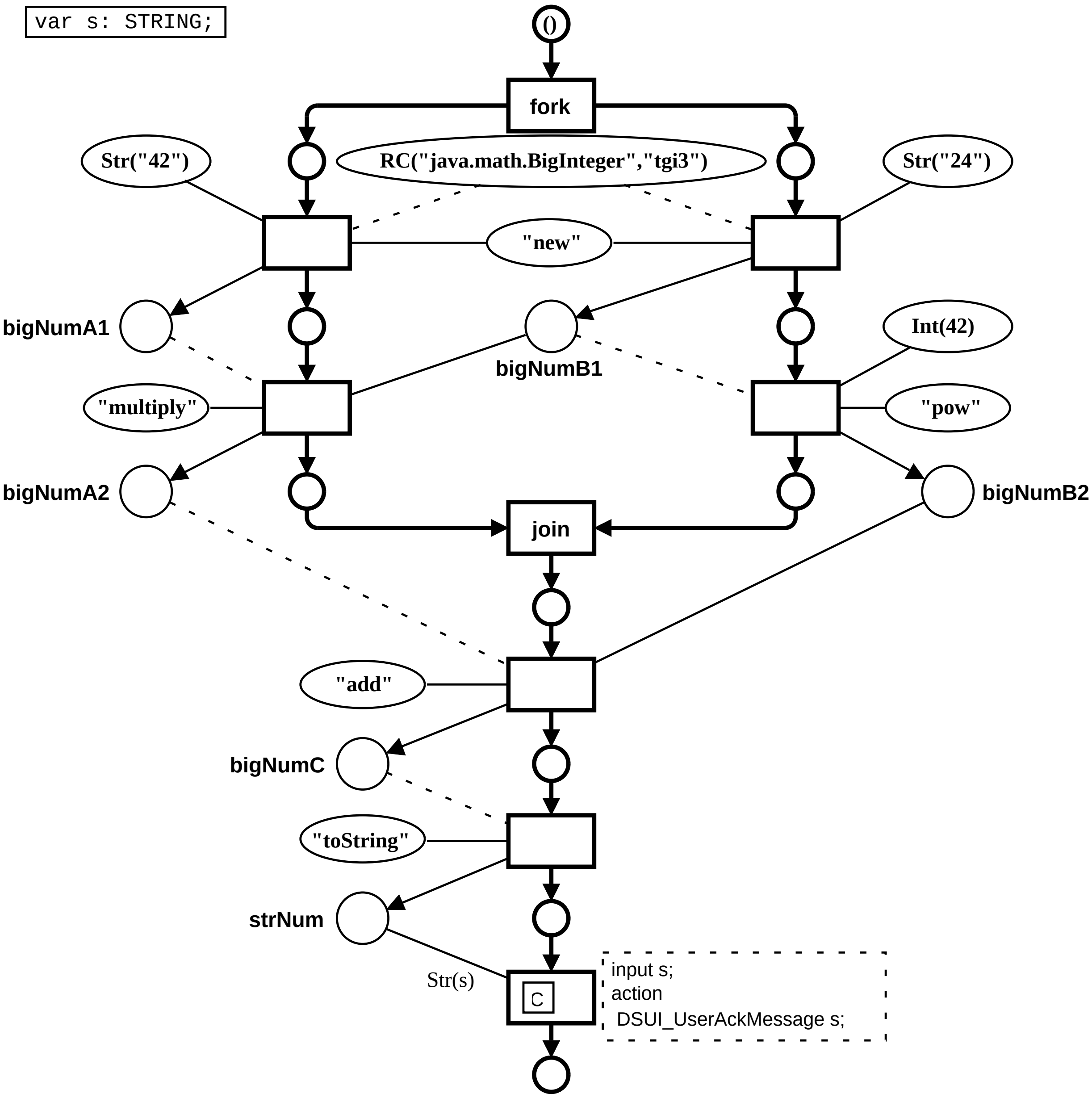


Figure 8: A Petri net performing concurrent calculations with Java `BigIntegers`

### 3.5 An Example Net Calling Java

In the following example net some more features of Java-calling Petri nets are illustrated. Why would we take the overhead of constructing a Petri net to call Java methods? Two main reasons are that Petri nets have a graphical representation (thus, we have something like a visual programming front-end for Java) and that they are superior in specifying concurrency.

In the example in figure 8, again a computation with big integers is performed, but this time, concurrency is exploited. To specify concurrency, we use transitions to fork and to join control flow. Since both "threads" that are created send messages to Java, we have to take care that the return messages are correctly associated. This is ensured by every instance of the `Call` subpage using a different identifier.

There have also been other test applications. In [4] a game was developed that simulates a stock exchange using a Coloured Petri Nets. The net was augmented by calls to a graphical user interface programmed in Java, resulting in a game that can be played by multiple players across a network. The new net shows a dramatic increase in usability that would have been impossible with other tools.

### 3.6 Processing Method Calls in Java

We now briefly describe how the Java side of our framework treats the incoming messages. As a first step, a message is converted into an internal representation using a straightforward top-down down-parser.

Now some remote references might point to an object of the local Java process. Class references are immediately resolved to ordinary Java classes.[4] Then the framework has to translate the remote instance references into local Java objects using a special table of externally known objects.

Afterwards the framework determines the class of the invoked object and uses the standard `reflection`[5] package to get a list of all public methods which the object supports. It chooses the appropriate method from the list and calls it in an individual thread. Now other calls can be concurrently received and parsed.

After the completion of the method call, the result is sent back to the caller whose address was specified in the message. If a reference to a Java object is returned, the framework generates a unique external name for it and inserts this name in the table of externally known objects. Design/CPN will then receive the message and forward the result.

If a Design/CPN process interacts with multiple Java processes, remote references might point to an object of a remote Java process instead of a local Java object. This case, too, is handled by our framework, e.g. one could, if desired, store a reference to an object on one process in a hashtable on another process.

There are two possibilities for a CPN process to get to know Java objects. To start with, such an object may be created by a special program that also invokes the framework and inserts the object into the table of externally known objects under a well-known name.

Then, the CPN process may access the `new` method of a class reference. In section 3.2 we have given an example of such a call. In fact, the message looks like a normal method call, which is not the case in the Java programming language, where `new` is a keyword with a special syntax. To keep the message format clean, we decided to use `new` like a static method provided by every class instead.

This allows the creation of arbitrary Java objects from a Design/CPN net, even for built-in classes like hashtables, windows, etc. Although this device is extremely powerful, a note of caution has to be: The creation of new objects using the `new` method provides complete access to the Java environment, thereby opening up huge security holes. But Java programs can deliberately reduce their access rights by means of the `SecurityManager` interface. If this does not prove sufficient, it is still possible to protect the access to either the Java framework or to the entire message handling system by passwords or other techniques.

[4]Java allows the loading of classes at runtime requiring only a string representation of the class name. Even the creation of classes at runtime is possible.

[5]The `reflection` mechanism is a powerful feature of the Java environment that allows the complete analysis of objects, classes and methods at runtime. Thus, Java programs can view a *reflected* image of themselves as if in a mirror. Additionally, the modification, the creation, and the invocation of objects is supported.

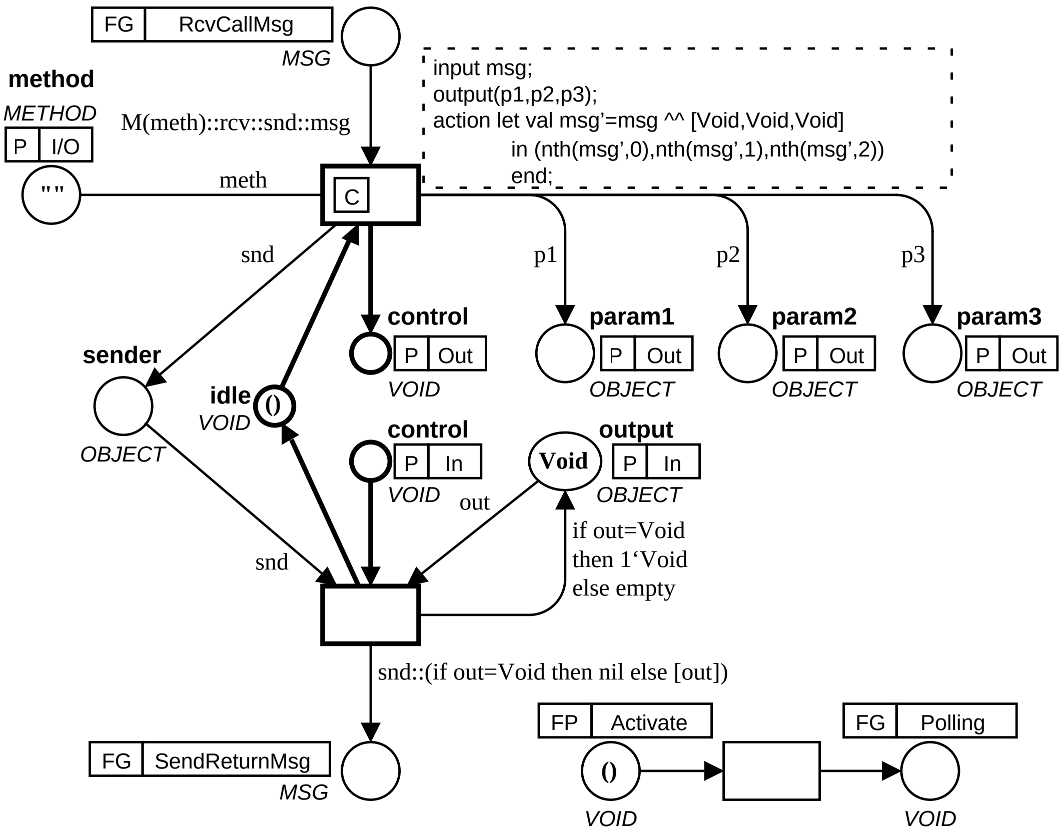


Figure 9: A subpage for calls to a CPN (`CPNCall`)

# 4 Accessing Design/CPN from Java

In the last section, we have shown how to call Java from Design/CPN. This approach is useful when the major part of the application is modeled with Design/CPN and Java is used to complement the application.

In other cases, an application or at least large parts may have already been implemented in Java, but Petri nets are utilized for modelling use cases or workflows that are contained in the application. Then, one may wish to design and run these parts with Design/CPN and call Petri nets from Java. The basic idea is that a system modeler can use the most appropriate tool for representing and solving the problem at hand.

All one has to do to extend our approach to support this feature, is firstly to implement generic Java proxy objects that call Design/CPN nets via messages. Secondly, Design/CPN nets have to be extended to be able to receive and process call messages, not only return messages. In a sense, a Design/CPN process has to behave like a single (static) object, thus providing some methods that may be called.

## 4.1 Designing a Subpage for Calls to Design/CPN Nets

Again, we have tried to find a very general solution, i.e. a net subpage that can be re-used for any Design/CPN net implementing some method call. Figure 9 shows such a subpage which is basically the counterpart to the net presented in figure 6. The two transitions at the bottom ensure that the net is able to receive messages all the time, in contrast to a net that just calls Java methods and polls for return messages if some call has been sent.

The upper transition checks all incoming messages whether they are call messages to the method name given in the port place `method`. If so, the parameters contained in the call message are distributed into separate places (again, the maximum number of parameters is restricted to three) and a token is put into the `control` output port place to specify that the "parent" net may start now. The receiver of the message is ignored, because the whole net is treated like a single object. If this net had not been the receiver of the message, the message would not have been delivered there. The `idle` place is needed because this simple version of calling a net does not support concurrent calls to the same net, while it does allow concurrency within the net. This restriction is lifted when our object Petri net approach is used as discussed in the conclusion. After the parent net has finished its task, it has to put a token into the `control` input port place. It should remove all input parameter tokens as well as all tokens that were produced during the run, lest the next call fails or produces unexpected results.[6] When a socket place is assigned to the `output` place, some token has to be present that is used as a return parameter. The subpage then constructs a return message to the sender of the call message and puts it into the global fusion place `SendReturnMsg`, so that it is sent by the `Message` page and the `IOPage`.

### 4.2 An Example of a Net that can be Called

To illustrate the use of this subpage, we specify a net that implements a method executing some example workflow by W. v.d. Aalst, cited in [14].

The example was introduced as follows. When a criminal offence happens and the police has a suspect, a record is made by an official. This is printed and sent to the secretary of the Justice Department. Extra information about the history of the suspect and some data from the local government are supplied and completed by a second official. Meanwhile the information on the official record is verified by a secretary. When these activities are completed, a prosecutor determines whether the suspect is summoned or charged, or whether the case is suspended.

In figure 10, a workflow for this case is modeled using our object-oriented Petri net notation introduced in section 3.4, extended by the feature that an invocation transition may consume input parameters. In the upper part, the interface places of the method `crimeCase` are connected to the subpage `CPNCall` through a substitution transition. The arcs that point to the bottom border of the figure are actually connected to this transition, too. Note that tokens for the places `official1`, `official2`, `secretary`, and `prosecutor` have to be provided elsewhere. We do not give any further details on how the invocation transitions might be refined. In fact, the activities may be implemented as code regions, as subpages, in Java or even as other nets that may be called through the message interface.

In this case the call of the workflow from Java simply looks like

```
RemoteCPN cpn=new RemoteCPN("cpnhost:12345");
String decision=(String)cpn.execute("crimeCase","Roger Rabbit",
                                    "murder of Marvin Acme");
```

assuming that `cpnhost:12345` is the hostname / process-ID of the Design/ML process (and thus its mailbox name) simulating the workflow net, that the suspect is Roger Rabbit, and that he is accused of murder, where the decision of the prosecutor (as well as the input parameters) is implemented as a `String`.

[6] It is not too complicated to check this property automatically using place invariants.

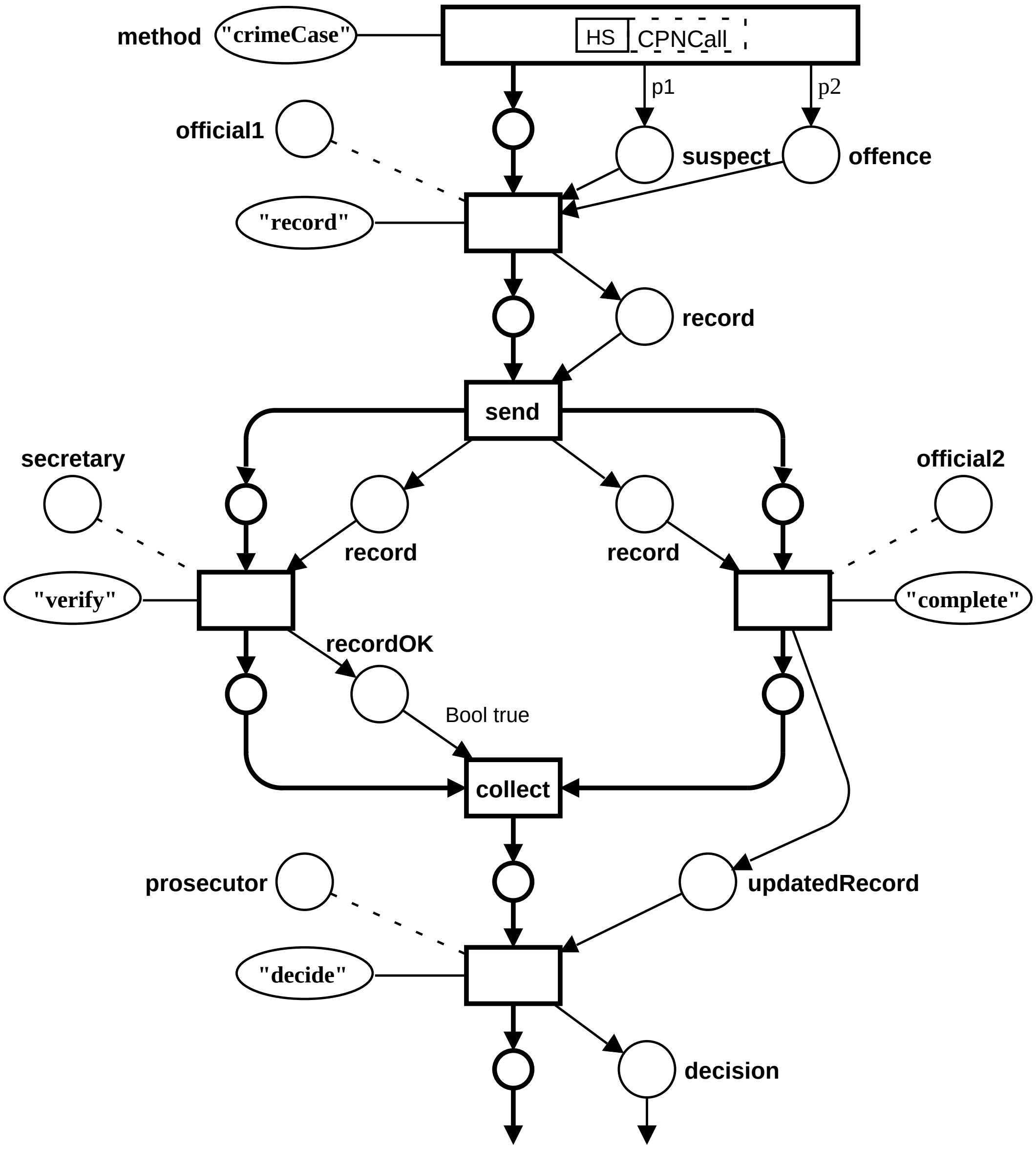


Figure 10: An example of a net that can be called from Java

# 5 Possible Applications and Benefits

Petri nets are already a concurrent formalism, so a net model documents the intended concurrency without the need to distribute it. But there remain at least two cases where a distributed net model is favourable even in the absence of interaction with Java:

- Need for performance gains. Whether any significant performance improvements are possible depends on several factors. It must be possible to split the application into parts that communicate by messages. This is often naturally the case, but existing net models might not show the possible splitting lines clearly. What is more important is that the nets should require as little communication messages as possible. Alternatively, the net might be demanding in terms of ML evaluation time, e.g. animation or optimization algorithms. In both cases, the communication cost might be dominated by the computation cost.

- Real concurrency. The Design/CPN simulator is a single-threaded application. Thus, no real concurrency, not even multi-threading is available. Multi-threading would be especially desirable if complex or time-consuming ML functions are executed, which is normally done within one simulation step, delaying all transitions. Now, code may be moved to a separate diagram (or any external program, see below), enabling the calling net to continue working while the computation is performed. The drawback of polling for the answer is in some cases preferable compared to blocking the whole simulation.

- Necessary distribution. Some applications require access to distributed resources, e.g. a visualization module that accesses various screens or a game that needs input from many players. Such applications cannot naturally be realized without a distributed simulator.

However, the greatest benefit of our communication framework comes from the interoperability with other programming environments:

- The access to Java processes enables the developer of a net model to incorporate much more complex GUIs into a net. This improves the interaction with the user of the net, but might also be used to animate and visualize the simulated process or to generate more elaborate statistics and debugging information.

  Although there are some GUI libraries for Design/CPN already—the most notable one being Mimic/CPN described in [13]—they do not match the flexibility of Java or comparable languages. Moreover, there is a lack of rapid prototyping tools which greatly speed up the GUI development.

- Distributed computing also allows multiple users to participate in a single simulation from different terminals.

- Processes that are controlled by Design/CPN might also handle general I/O devices. This may simplify the control OF a system by a net, a possibility already mentioned in the context of the security system presented in section 1.5 of volume 3 of [7]. There it was proposed to extract parts of the ML code generated from the net for the execution on a stand-alone microprocessor. Such a task might be considerably simplified if the connection to the outside world remains the same during the translation.

- It becomes possible to reuse code that was not developed in Design/CPN or ML. In this area the standard container classes come to mind immediately, but in fact there is a wide range of programs for Java covering almost all aspects of algorithms and data structures as well as various I/O and network libraries.

There are additional benefits when Coloures Petri Nets can also be called from the outside:

- It is possible to move gradually from a Petri net prototype to an implementation using Java. Although this will require that the nets themselves are structured in an object-oriented way, it remains a viable route.

- The specification of workflows with Petri nets has attracted much interest in the past years. Using the framework it seems possible to use Coloured Petri Nets for executable prototypes of workflows and in the future maybe even for final products.

## 6 Outlook

The framework presented here improves the usability of Design/CPN in some of the most important areas: distribution, interoperability, appropriate modelling, and graphical user interfaces. A smooth transition from a specification within Design/CPN via distributed Design/CPN modules to distributed Java modules seems possible. Some applications have already been implemented thereby documenting the gained flexibility.

In the near future we are going to extend the framework to object-oriented Petri nets in the sense of [8], [11], and [10]. In those approaches the structure of the nets represents most object-oriented features without extensions of Coloured Petri Nets themselves. Tool support of the object-oriented techniques could then further simplify the development process.

Additionally, Artificial Intelligence concepts as already used in [12] are going to be extended by providing a connection to Prolog. This will allow us to use available tools for logic programming directly from Design/CPN models.

If Design/CPN itself was extended by multi-threaded simulation of net-models, this would capture some concurrency aspects of our framework. The distribution and interaction with Java would still be essential benefits of our framework, which would even be improved, because blocking input could be used instead of polling.

Our communication framework already bears some resemblance to other distributed object-oriented architectures. Although many functions and just as many concepts are still missing, it does no longer seem far fetched that Petri nets might one day be usable with systems like CORBA [2].

## 7 Acknowledgments

Much of the Java implementation described in this paper was done during a project at the University of Hamburg. We would like to thank the participants for their efforts, especially Matthias Ernst, Torben Kroll, Heiko Rölke, and Eberhard Wolff.

A message passing library in C was implemented by Heinrich Biallas and has been documented in [1].

In [4] Margret Freund-Breuer and Olaf Fricke developed the stock exchange model that was used as a first test case for our Java interaction framework.

## References

[1] Heinrich Biallas. Realisierung der verteilten Ausführung von gefärbten Petrinetzen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, November 1997.

[2] The OMG Corba Page. WWW page at `http://www.omg.org/corba/`. Contains references to the current specification of CORBA/IIOP (on 05-15-98, this is version 2.2), a discussion of the Object Management Architecture, and many more links.

[3] Design/CPN Online. WWW page at `http://www.daimi.aau.dk/designCPN/`.

[4] Margret Freund-Breuer and Olaf Fricke. Spezifikation mit gefärbten Petri-Netzen am Beispiel des Börsenspiels. Studienarbeit, Fachbereich Informatik, Universität Hamburg, September 1993.

[5] Dirk Hauschildt. A petri net implementation. Fachbereichsmitteilung FBI-HH-M-145/87, Universität Hamburg, Fachbereich Informatik, 1987.

[6] The Java Home Page. WWW page at `http://java.sun.com`. Contains references to online material as well as to many introductory books and technical papers.

[7] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992: Vol. 1, 1994: Vol. 2, 1997: Vol. 3.

[8] Christoph Maier and Daniel Moldt. Objektorientierte Konzepte – Dargestellt mit gefärbten Petrinetzen. Fachbereichsbericht FBI-HH-M-261/96, Universität Hamburg, Fachbereich Informatik, August 1996.

[9] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Handbook Version 2.0*, 1993.

[10] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen*. Dissertation, Universität Hamburg, Fachbereich Informatik, August 1996.

[11] Daniel Moldt and Christoph Maier. Coloured Object Petri Nets - A Formal Technique for Object Oriented Modelling. In B. Farwer, D. Moldt, and M.-O. Stehr, editors, *Petri Nets in System Engineering, Modelling, Verification and Validation*, pages 11–19, Fachbereich Informatik, Univ. Hamburg, 1997. FBI-HH-B-205/97.

[12] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured petri nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, number 1248 in Lecture Notes in Computer Science, pages 82–101, Berlin, 1997. Springer Verlag.

[13] Jens Linneberg Rasmussen and Mejar Singh. *Mimic/CPN – A Graphics Animation Utility for Design/CPN. User's Manual Version 1.5.* Computer Science Department, Aarhus University, December 1995.

[14] Rüdiger Valk. On Concurrency in Communicating Object Nets, 1998. Accepted for publication at the International Conference on Application and Theory of Petri Nets (ICATPN).