# Tight Integration of Java and Petri Nets

Olaf Kummer

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

**Abstract**

In this paper we investigate how the Java programming language can be integrated in an object-oriented Petri net formalism. It becomes apparent how a carefully designed unification algorithm supports this goal, despite the fact that Java itself is not based on unification. Some dedicated algorithms for a Petri net simulator are proposed.

## 1 Introduction

The basic net formalism discussed in this paper are reference nets. In reference nets, a net can be instantiated arbitrarily often and references to net instances can be used as tokens in other net instances. For details see [6]. Reference nets use a variant of Java as their inscription language. See [2] for the original Java specification.

This paper can be seen as a companion to the earlier work [4], where the reference net formalism and the large scale architecture of the Petri net simulator Renew are described. Here we will discuss some further aspects that were only very briefly sketched in [4].

## 2 Tuples

The Java language has no support for tuples. The only case where tuples would come in handy in Java would be multi-value returns. However, these occur not very often and can be substituted by the return of a complex object that aggregates the return values. In other cases, when we have multiple values that should be processed together, we can simply use multiple variables that hold one value each.

But for Petri nets, tuples are more important. It is dangerous to store related values in multiple places, because, if multiple tokens have to be put there, tokens for different data sets can be mixed. Tuples solve this problem.

Java's syntax offers two opportunities for adding tuples. Either a tuple is written `(1,2,3)` or `[1,2,3]`. The first form has the disadvantage that it is impossible to write 1-tuples, because they could be mistaken for grouping operations, e.g. `"string"+(true|false)` would be ambiguous. Hence it was decided to use the square bracket notation. However, when we want to support lists as well as tuples, we have to fill *both* syntactic niches. In that case, 1-tuples might be considered dispensable, but lists of length 1 would be mandatory, so that the decision would have been the other way round.

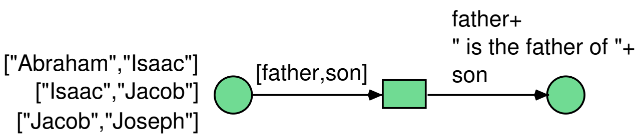


Figure 1: The net `tuples`

As we have tuples in the languages, it seems sensible to provide a pattern matching capability and thus unification. In Fig. 1 you can see a simple example net. Each token in the place can be unified with the input arc inscription to assign values to the variables `father` and `son`.

It was decided to implement an occurrence check in the unification algorithm, i.e., a tuple must not contain itself. Due to efficiency considerations, this check is often omitted. But this might lead to undesirable behavior, especially dead loops, which was considered unacceptable.

Besides tuples, the unification algorithm deals with variables and unknown objects. Variables can hold a value, although they are not supposed to be used as values themselves. Unknown objects are the initial value of variables. They represent the identity of values when the values themselves are not yet known. Consider for example two unassigned variables `x` and `y`. After unification, both will reference the same unknown object as their value. After unifying `x` with `1`, also `y` will be valued with `1`.

The unification algorithm must also unify ordinary Java values, which can appear as components in tuples or as values of variables. Here unifiablility could be based on identity or on equality. Java provides a method `equals(...)` in its object model that is supposed to test objects for equality and an operator `==` that checks identity. Identity is safe in the sense that it is a constant equivalence. On the other hand, equality is *supposed* to be an equivalence, but this cannot be enforced, and the equality can change over time.

Nevertheless, we chose equality as the basis of unification. Firstly, it is not even guaranteed that even two equal text strings are identical, so that unexpected results could occur. Secondly, all existing container libraries are based on equality. As we had to use lots of container classes in the design, we had to rely on the functionality of `equals(...)` in any case. It turned out that the resulting problems are comparably minor, but still care has to be taken.

The unification algorithm works in-place, i.e., the original tuples are modified, if necessary. Similarly, the contents of variables may be changed by a unification. This way we can avoid excessive copying during each unification. Of course, we have to add a backtracking mechanism to undo unifications, because the state before the unification is destroyed. Backtracking is invoked whenever a proposed binding of variables is discarded by the simulation algorithm.

All modifications to unifiable object like tuples and variables are collected in a so-called state recorder. It is possible to have multiple state recorders that can concurrently record and undo changes in different unifiable objects. However, the unifications in a single unifiable objects have to be undone in the correct order.

Pattern matching is useful for input arcs inscribed with tuples. Here one component of the tuple might already be known to the Petri net simulator, and there might be very few tuples in the place that match this value compared to the number of all tokens. Therefore indexes allow a quick lookup of partially specified tuples. However, due to performance considerations only single tuple components are used as indexes.

If all combinations of tuple components were maintained as indexes, a combinatorial explosion would result. Of course this means that arc inscriptions like `["Joe","Average",phone]` do not allow an extremely efficient lookup of Joe Average's phone number in the place that contains all phone numbers, because two components would have to be respected. However, the simulator will still limit its search to all Joes or all Averages, whichever constitutes the smaller set. To help the simulator, you could use nested tuples for storing the data. Now, `[["Joe","Average"],phone]` does give you a constant time lookup, because the first component of the tuple, a pair actually, is completely known and forms a valid index.

A similar indexing algorithm might be contained in several other simulators, but literature on this topic is very sparse. Only [3] gives information on a related approach, which is somewhat less flexible and not as efficient as the proposed scheme, however.

## 3 Actions

Many high-level Petri net formalisms support the execution of some additional code during the firing of a transition that does not influence the enabledness of a transition. Reference nets support this concept by so-called action inscriptions, which can trigger side effects, but may also be used to determine the value of output arc variables. A common use would be the transition inscription `action x=y.aLongMethodCall(u,v)` to initiate a long method call that has no influence on the activation of the transition.

Because the result of an action is unknowable, we must make sure that every result will lead to a legal firing of the transition. The simulator should detect the following illegal situations:

- A variable is calculated by an action but is already computed during the search for a binding.
- A variable is calculated by two different actions.

- A cyclic dependency between the actions exists where each action requires a value that has to be computed by a different action.

Luckily, we can integrate the detection algorithm into the already implemented unification algorithm. We represent an action call by a calculator object that is unifiable only with itself. This object references its arguments, which might be Java objects, tuples, unknown, or even other calculator objects. The occurrence check can thus detect even those cycles that arise out of recursively embedded tuples and calculator objects.

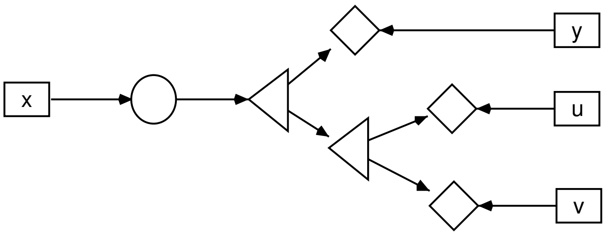


Figure 2: A data structure of unifiable objects

Fig. 2 visualizes the actual data structure generated by `action x=y.aLongMethodCall(u,v)`. The variables are denoted by rectangles, the calculator object by a circle, the unknowns by rhombi. There are two tuple objects depicted as triangles that result from the internal representation of the expressions. One tuple groups the arguments and one tuple combines the target object and the other tuple. This simplifies some internal procedures.

At this point, it would be impossible to unify `x` and `y`, because a cycle would arise. It would be impossible to unify `x` with `1`, because the calculator object can only be unified with itself.

Another part of the unification algorithm detects when a calculated value is required for the evaluation of an input arc. In that case the transition cannot be enabled, because the value is required early to see if an appropriate token is available, but on the other hand, the value must not be computed early, because it results from an action.

But what happens during the firing of the transition? A value must be assigned to `x`, therefore the variable must no longer be bound to a calculator object. The removal of the calculator objects is done immediately after a successful search. Imagine that the search algorithm has found a valid binding. An output arc expression has been computed, resulting in a tuple. We cannot put that tuple directly into the place, because the tuple's content might be subject to backtracking. Therefore we make a copy of all variables and tuples and ensure that the copies are immutable. During this process, which has to be done in any case, we can safely remove all calculator objects without further inefficiencies. They served their purpose of ensuring uniquely calculated values and are no longer needed.

# 4 Calling Nets from Java

In [5] an architecture was discussed that allows the integration of Java and Petri nets that are implemented with the Design/CPN tool. Using Java-inscribed Petri nets we can provide an even smother transition between the two worlds. In fact we have already seen that calls to Java pose no problems even in the presence of side effects when we use action inscriptions.

When we want to call nets, we must define methods for nets. The wish to implement net methods is not new and plenty of schemes have already been devised, [7] could be named as an example.

In reference nets, the only means of communication with nets are synchronous channels. Channels are parameterized and allow a bidirectional information transfer. In order to synchronize, two transition instances have to agree on a channel and on a sequence of matching arguments. The transition instance that initiates the synchronization must explicitly reference the net instance where the target transition is located. E.g., `net:channel(2,3,5)` would be an invocation through the channel `channel` of a transition in the net `net` with three parameters. The target transition would have to provide an inscription like `:channel(x,y,z)` to signal that it wants to engage in synchronizations through that channel. This concept is essentially based on the channels proposed in [1].

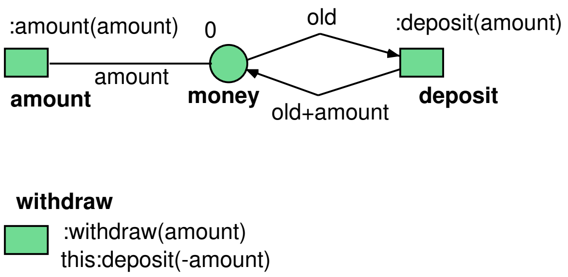


Figure 3: The net `account`

The net from Fig. 3 models a very simple bank account. The customer can only deposit and withdraw money and view the current amount. What we need now is a wrapper class that converts method calls to channel invocations, so that we can use the net as an ordinary Java object. We have describe how the conversion is done.

```
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 `samples.call` is given in a special statement. A compiler can now be invoked that converts the description into Java code. The resulting class `Account` will be known as a *stub class*.

Each time you generate a stub object, the associated net is automatically created, too. This is the reason for listing the net name after the keywords `for net` in the header.

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.

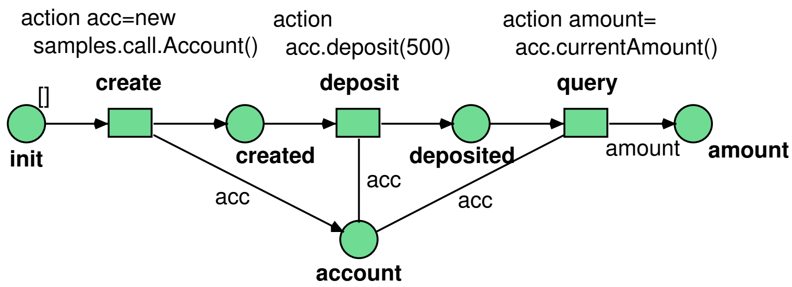


Figure 4: The net `customer`

The net from Fig. 4 exemplifies the use of the stub class. During the execution of a method, the invoking transition is blocked and cannot yet process its output arcs. The stub object provides a synchronization request to the search algorithm. The search algorithm can then try to find an appropriate synchronization or fire additional spontaneous transitions in the case that the synchronization is not possible at the moment. After the synchronization succeeds, the stub object is informed, so that the method can return.

It is important to note that a method call can consist of multiple synchronization requests, so that you are not limited to actions that can be performed atomically by a single transition. It is allowed to start large subprocesses during the execution of a net method. Typically, exactly two synchronizations are required, one to start the method, and one to collect he results, but this may vary. A common method description might be

```
type method(type0 arg0, type1 arg1, type2 arg2) {
  this:method(instance,arg0,arg1,arg2);
  this:result(instance,return);
}
```

Here we provide an additional argument `instance` that is set by the invoked net during the first synchronization. It can be used during the second synchronization to collect the correct return value just in case multiple method calls have been made. We do not specify the type of `instance`, but it might be a unique call number or a reference to a net instance that is created to process the request.

## 5 Types

Although in the application domain of rapid prototyping it is desirable to have a type free system where variables do not have to be declared, it is also useful to have the security of type checking, if desired. Therefore reference nets provide both a typed and an untyped formalism. Not only variables, but also places may be typed. In both cases a type guarantees that only values of that type may be stored in the typed object.

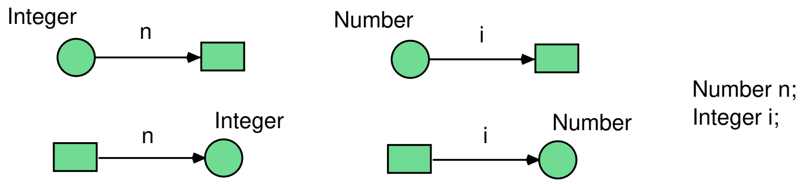


Figure 5: The net `types`

The net from Fig. 5 illustrates a key problem while implementing a type system. Which conversions should be allowed for input and output arcs? If the arc type and the place type are neither supertypes nor subtypes of each other, is seems plausible to report an error. If the developer knows that all transported values belong to both types, this should have been indicated with a cast.

If the type of the output arc inscription is a subtype of the place type, everything is correct. But if it is not, we cannot be sure that the token that is put in the place satisfies the type constraint. Therefore we have to report an error for the lower left transition in the figure, because a number is not necessarily an integer.

Perhaps surprisingly, we allow both supertype and subtype relations for input arcs. If the result of an input arc expression turns out to be of the wrong type, it is guaranteed that no appropriate token is in the place, so that this binding is not enabled anyways. On the other hand, if the input arc is inscribed with a subtype of the place type, we simply have to make sure that only those values that are permissible are unified with the variable. Other tokens in the place are simply ignored.

The Java type system is relatively straightforward, but it has a few rough edges that stem from Java's origin in C++. Especially, Java will gladly convert 32-bit integers to 32-bit floating point numbers or vice versa, although information is lost in the process.

This is acceptable, albeit dangerous, for ordinary assignments or method calls. For token removals such lossy conversions should not be done automatically, because in a Petri net, the direction of information flow is not always obvious. E.g., a float place might have an outgoing arc inscribed with an integer variable. It is not clear whether we should accept the loss that results from converting float tokens to integer values when assigning the variable or the loss that results from converting the integer value to a float, so that the token can be removed.

Therefore a new subtype relation *losslessly convertible* was introduced into the system. It is only used for type checking arcs and it differs from the normal subtyping relation by not allowing conversions from longs to either floats or doubles and from integers to floats. For method calls the ordinary Java semantics was retained to achieve compatibility.

Synchronous channels are untyped. Because the direction of information transfer is unspecified for channels, the concept of lossless convertibility will be used here, too, if it is decided to add typed channels in the future.

At the moment, tuples have one common type, as we do not differentiate between integer triples, string pairs, tuples of arbitrary objects, and so on. Typed tuples would be a valuable addition, but at the moment there are too many conflicting approaches, none of which is supported by the Java environment. These issues should probably be postponed until it has been finally decided whether and how to integrate parameterized types in the Java language.

# 6 Conclusion

We have seen how a proper integration of Java and Petri nets leads to various design decisions on both the net formalism and its implementation. Starting with tuples we required unification and pattern matching. We saw that action inscriptions, which greatly add to expressiveness, gave rise to a surprising extension of the unification algorithm. The wish for a seamless integration motivated calls from Java to nets, which are supported by automatically generated stub objects. All algorithms have been implemented in Renew 1.1, which is freely available [6].

# References


[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] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.

[3] Torben Bisgaard Haagh and Tommy Rudmose Hansen. Optimising a coloured Petri net simulator. Master's thesis, University of Aarhus, December 1994.

Available at `http://www.daimi.au.dk/CPnets/publ/thesis/HanHaa1994.pdf`.

[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*, Forschungsbericht Nr. 694, pages 73–78. Fachbereich Informatik, Universität Dortmund, October 1998.

[5] Olaf Kummer, Daniel Moldt, and Frank Wienberg. Symmetric communication between coloured Petri net simulations and Java-processes. In Susanna Donatelli and Jetty Kleijn, editors, *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 86–105. Springer-Verlag, 1999.

[6] Renew – the reference net workshop. WWW page at `http://www.renew.de/`. Contains the documentation for Renew and an introduction to reference nets.

[7] C. Sibertin-Blanc. Cooperative nets. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490. Springer-Verlag, 1994.