Animating Formal Specifications Using Java Applets

Charles Lakos and Glenn Lewis
Electrical Engineering and Computer Science
University of Tasmania
GPO Box 252-65
Hobart, TAS, 7001, Australia.
Charles.Lakos@utas.edu.au
Glenn_Lewis@eeecs.utas.edu.au

Abstract
This paper proposes the generation of Java applets from specifications given in an extended Petri Net formalism. The anticipated advantages of this approach are to gain a greater confidence in the correctness of the final product, and to harness the capabilities of Java in a consistent framework. The emphasis of this paper is on the generation of the Java code from the Petri Net specification, on the assessment of both Java and C++ as suitable targets for this translation, and on the suitability of this extended form of Petri Nets as a host for various extensions to Java.

1 Introduction

Since the initial release of Java in May 1995, there has been an explosive growth in its use and in the available functionality. This explosive growth has been fuelled by the incorporation of Java into web browsers, the use of Java as the foundation for network computers (released in October 1996), not to mention the extensive advertising hype. There are numerous extensions to Java (some released initially, and some only appearing more recently), including multithreading, persistence, database interface, native code interface, distribution, client-server support, component support, revised GUI.

There are significant advantages in having this wide range of functionality available in a single, platform-independent language, but it raises the question as to whether this functionality can be effectively harnessed. One proposal for simplifying the construction of Java applets is the use of Java beans, which provide a simple component software approach. However, it is not clear whether this approach will have any significant impact on the notoriously difficult task of writing and debugging multithreaded or distributed applications.

This paper considers the possibility of harnessing the functionality of Java by generating applets which animate formal specifications. This is part of an ongoing project centred on the formalism of Object Petri Nets (OPNs), which attempts to build on the proven benefits of Petri Nets — their graphical representation which can capture the interaction patterns of dynamic, concurrent systems; and the availability of analysis techniques which have been developed for nets. OPNs are distinctive (compared to other net formalisms) in using object-orientation to augment nets with compositional and incremental development capabilities.

We envisage an integrated environment based on OPNs which supports the following software development process:
1. Use some methodology (such as Shlaer-Mellor [19]) to analyse the system and produce a (static) information model and the appropriate dynamic models.

2. Build an Object Petri Net specification of the system based on the analysis of step 1, e.g. by using the approach presented in [11].

3. Generate a prototype Java applet from the formal specification of step 2. This applet can be run to simulate the system and get an intuitive feel for its behaviour. This is often a useful first step in understanding the system [1, 7]. This step will probably result in changes to the analysis model (step 1) and the formal specification (step 2).

4. Analyse the (modified) formal specification for properties such as deadlock and liveness, using algorithms developed for Petri Nets [6].

5. Embellish the net and thence the applet with an appropriate graphical user interface. This process will result in a formal specification (from steps 2 and 4) and a prototype Java applet (from steps 3 and 5).

There are a number of issues about this proposed development process which are not considered in this paper. The analysis of step 1 and the preparation of an OPN model in step 2 are not considered, though this has been addressed elsewhere [11]. The analysis of OPNs (step 4) is the subject of current research. We anticipate that, like the analysis of other high level Petri Net formalisms, the state space explosion problem will make it necessary to restrict the analysis to only part of the system or at least a reduced form of the system [1, 7].

Another matter which is not considered here is the necessity of optimising or even hand-coding the application or prototype once the formal specification has been validated. This will clearly be driven by the demands of efficiency, which will consequently be dependent on the particular application. However, any modification of the code generated from the specification will introduce a possible source of errors. We anticipate that a computationally-intensive application will require optimisation, but it is not clear that there is much interest in coding such applications for Java. On the other hand, applications which involve low computation demands and are largely limited by user interaction speed and network latency, will probably not require optimisation.

The focus of this paper is on step 3 — the generation of Java applets from formal Petri Net specifications. A more extensive treatment of this work may be found elsewhere [16]. The implementation of this translation is discussed, though it is still being refined and extended. An interesting question is the extent to which Petri Nets provide an appropriate paradigm for harnessing the extended features of Java. Given that Petri Nets naturally embody concurrency, we anticipate that multithreading can be incorporated by annotating the net to indicate which parts are to be run as separate threads. The semantics of the net doesn’t change. Similarly, the simple interaction between net components in terms of token transfer maps naturally into message passing, thus laying the foundation for a distributed implementation of the nets. Petri Nets also have a natural notion of data repository, thus suggesting that persistence can be harnessed in a natural way. These extensions (together with others) are considered briefly in §6.

The paper is structured as follows: the form of OPNs currently implemented is briefly summarised in §2; some examples illustrating the current capabilities of the system are presented in §3; the main aspects of the mapping from OPNs to Java are considered in §4, and an assessment of the suitability of both Java and C++ as targets for this translation are given in §5; a consideration of how the more recent Java extensions can be incorporated into a uniform framework is given in §6, and the conclusions are presented in §7.

2 Overview of Object Petri Nets (OPNs)

This section reviews the formalism of Object Petri Nets (OPNs), which are an enhanced form of Coloured Petri Nets (CPNs), which in turn are an enhanced form of elementary nets or Place
Transition Nets (PTNs).

2.1 PTNs: The four quarters of the year

A simple example to demonstrate the concepts of Petri Nets is that of the four seasons or the four quarters (of the year). This can be modelled as the Place-Transition Net (PTN) or simply the Petri Net (PN) of fig 2.1 [18].

![Fig 2.1: The four seasons or four quarters](image)

This PTN has four places drawn as circles representing the four quarters of the year (and labelled $q_1$, $q_2$, $q_3$, $q_4$), and four transitions drawn as squares representing the changes between the quarters (and labelled $n_1$, $n_2$, $n_3$, $n_4$ to indicate the next quarter). The arcs are also labelled ($b_1$, $a_1$, etc.) to indicate the before and after state of each change. The initial marking indicates the presence of a token in place $q_1$, implying that we begin in the first quarter. In this initial marking, only transition $n_2$ is enabled, and when it is fired, the token is removed from place $q_1$ and a token is added to $q_2$.

The simplicity and elegance of Petri Nets arises from an even-handed treatment of state and change of state (since each is represented by a node in the graphical notation). This leads to the ready modelling of non-determinism and true concurrency. For example, if the net of fig 2.1 had a token on place $q_3$ as well as on $q_1$, then a single step of the net could include the concurrent firing of transitions $n_2$ and $n_4$.

![Fig 2.2: Simple airport net](image)

Another illustration of this might be the modified net of fig 2.2 which represents an airport where arriving planes are indicated by the tokens in place $arr$. (These may be present initially, or deposited by some other subnet.) These planes circulate around the four quadrants of the...
airport in a holding pattern until they can land from the first quadrant (place \( q1 \)). In order to start landing, the runway needs to be available, as indicated by a token in the place \( free \). Once the plane has landed, a token will be added to the place \( shed \) and a token will be returned to the place \( free \), thus allowing another plane to land. Clearly, this example can be extended to multiple arriving planes. Then, the progression of one plane from quadrant 2 (place \( q2 \)) to quadrant 3 (place \( q3 \)) is independent of the progression of another plane from quadrant 4 (place \( q4 \)) to quadrant 1 (place \( q1 \)). Petri Net semantics allows these two actions to occur concurrently. On the other hand, a plane in quadrant 1 may continue to quadrant 2 or start the landing procedure. In this case, the two actions are in conflict and only one can occur.

Extensive theoretical results and analysis techniques applicable to concurrent systems have been developed for Petri Nets. However, the above formalism lacks descriptive power for the convenient modelling of more complex systems.

2.2 CPNs: Including data values

In order to extend the PTN of fig 2.1 to indicate the year as well as the quarter, catering for \( y \) years would require \( 4y \) places and \( 4y \) transitions in a spiral rather than circular pattern. The simple net structure would be lost, unless we move to a more expressive Petri Net formalism, such as that of Coloured Petri Nets (CPNs) [5]. A CPN can model the same extended system using the same structure of fig 2.1 simply by associating an integer colour with the token. Now the labelling of the arcs becomes significant, since this must indicate which coloured token is removed and added by each transition. Two possible styles for indicating this evolution are given in parts (a) and (b) of fig 2.3 for each of transitions \( n1 \) and \( n2 \). For transition \( n2 \) (the next quarter is 2), the year does not change. For the transition \( n1 \) (the next quarter is 1), the year does change. In part (a), distinct identifiers are associated with distinct arcs and the relationship between the associated tokens is given by the guard associated with the transition (enclosed in brackets). In part (b), matching expressions are used to relate the associated tokens. From a formal perspective, the two approaches are equivalent and we do not distinguish them further.

![Annotations for coloured tokens](image)

Fig 2.3: Annotations for coloured tokens

In a similar fashion, the net of fig 2.2 could be extended to a coloured net to model different kinds of aircraft and different altitudes (which would be required to avoid collisions).

2.3 OPNs: Including abstraction and objects

OPNs further enhance the descriptive power of Petri Nets by introducing the notion of objects. Thus, tokens, places, transitions and even nets are objects and are defined by classes. Further, we adopt the object-oriented practice of supporting polymorphism, i.e. of allowing subclass objects to appear in superclass contexts.

If we consider the example of fig 2.2, the tokens representing planes which circulate around
the net will be objects defined by a certain class, say some generic aircraft. They could equally well be objects defined by some subclass, such as propellor aircraft, jet aircraft, etc.

Similarly, each place representing the presence of a plane in a quadrant could be refined into a subnet which indicated the progression of the plane through the quadrant, as in fig 2.4.

![Quadrant: Plane*](image)

**Fig 2.4: Expanded detail of a quadrant**

Here, the label of the class frame indicates that the class name is Quadrant and it inherits from Plane*, i.e. a place holding Plane tokens. When a token is deposited by the environment, it is accepted by the put transition and added to the internal place flying. The transition progress may fire a number of times, in which case the token is updated to , to indicate progress through the quadrant. When the plane reaches the boundary of the quadrant, transition get will become enabled, thus allowing the token to be passed back to the environment. (Note that the guards which control when transitions progress and get can fire have been omitted in the interests of brevity.)

In the same way, a transition can be refined into a subnet to indicate more complex activity. Elsewhere, the conditions that should apply to such refinement of places and transitions have been considered [14]. This generalises and constrains the constructs provided by Hierarchical Coloured Petri Nets (HCPNs) [5].

Finally, we note that OPNs support the modelling of true hierarchical systems by having a single, unified class hierarchy [11]. Thus classes defining nets can be instantiated as tokens. In other words, even tokens are truly objects in that they can encapsulate both state and activity. Thus our aircraft tokens could encapsulate activity to flash lights, consume petrol, etc.

### 3 Examples

Traditionally, Petri Net tools have provided an integrated environment with a graphical editor, a simulator and analysis capabilities, e.g. Design/CPN [5]. This approach is helpful in deriving and analysing a model of a system, but is not conducive to producing a stand-alone prototype. The evolution of Java and the possibility of transporting applets across the Internet strengthens the case for the production of stand-alone prototypes. The development of tools for OPNs has therefore focussed on the compilation of net specifications into a widely-available object-oriented language, such as C++ or Java.

When an OPN specification is compiled by the system described in this paper, there is a range of possible forms of output. It is possible to set run-time flags so that the simulation of the net reports every action — the transfer of tokens to and from places, and the enabling and firing of transitions. Such trace output is usually too voluminous for normal purposes. Using the graphical capabilities of Java, it is also possible to produce an applet which presents a more meaningful form of output, namely a graphical representation of the net as in fig 3.1.

Note that this display has a number of components — a panel with controls for running the net (occupying approximately the top quarter of the display), a panel for the display of the net (occupying approximately the middle half of the display), and a panel for textual output (occupying...
approximately the bottom quarter of the display). The main program of the applet can selectively include each of these components. The textual output is produced by annotating transitions with appropriate function calls. Even without the above graphical display, these function calls can be used to produce textual output which is more selective and more descriptive than the built-in tracing capabilities already discussed.

![Applet Viewer: simpleairport_main.class](image)

**Fig 3.1: Display of simple airport applet**

Just as these function calls can be used to produce more meaningful textual output, they can also be used to produce more enhanced graphical user interfaces. As noted in step 5 of the proposed methodology in §1, it is possible to embellish the applet produced by the compilation of the net specification. Thus, the airport example of fig 2.2 can be refined as discussed in §2.3 and then embellished to produce the output of fig 3.2. In detail, this is achieved by modifying the main program to instantiate the net but not the graphical display of fig 3.1. The transitions are annotated with function calls to update the display to show the progression of planes in the various quadrants. (Note that this graphical display has been adapted from the sample applet available on the net at [http://hjs.geol.uib.no/html/java/sample44.html](http://hjs.geol.uib.no/html/java/sample44.html).)

Alternatively, the graphical image of the system can be shown alongside the display of the net, as in the example of the dining philosophers shown in fig 3.3. Again, the graphics have been adapted from a sample applet available on the net at [http://www.javasoft.com/hooked/CD/applets/DiningPhilosophers/index.html](http://www.javasoft.com/hooked/CD/applets/DiningPhilosophers/index.html).

The interesting thing about both these examples is that the controlling logic can be easily modified in the net without impacting on the graphical display. Thus, the dining philosophers
net currently reflects a system that cannot deadlock (since two forks are obtained at a time). But different controlling logic could easily be included by modifying this net.

Fig 3.2: Enhanced airport simulation

Fig 3.3: Dining philosophers example
4 Mapping OPNs into Java

Object Petri Nets (and their textual representation in LOOPN++) provide a significant advance over the earlier language LOOPN [9] with respect to the regularity of the language and the level of integration of object-orientation.

Nets specified in LOOPN were translated into the C language [8]. With the more complete integration of object-oriented ideas into LOOPN++, it is preferable to use C++ as the target [2]. More recently, the evolution of Java from C++ has prompted the use of Java [3]. In the process of developing the Java target, the translator has been modified to minimise the differences between the C++ and Java versions. This provides an interesting basis for a comparison of the two languages, particularly in the context of the claimed advantages of Java over C++ [4]. These comparisons are presented in §5. The material in this section is equally applicable to both the C++ and Java targets.

In order to simplify the generation of code from an OPN specification, a template-driven scheme has been adopted. Each construct from an OPN specification (or LOOPN++ program) is mapped into a separate clause in an intermediate representation. For each such clause, there is one or more coding templates, such as the one below. These templates provide code skeletons which can be tailored by parameterisation and by interaction with specific coding routines. The use of coding templates significantly reduces the effort in modifying the target code, particularly when a different target language is used.

For example, the template below may be used for generating the C++ code for a class. It is apparent that this includes text which can be generated as is. It also contains variable references such as `$CLASS$` which will be replaced by the appropriate value (in this case the name of the class) when the template is expanded. It also contains macro calls such as `$+featuredefn` and `$classdefn`, which result in the activation of coding routines (with these names) and thus allow more complicated coding decisions to be incorporated in the templates.

```c
#define CLASSBODY "\ 
/********************************************************** 
* File: ${CLASS}.C 
* Description: Class body for '${CLASS}' 
* History: Generated by LOOPN++ 
**********************************************************/\n
#include "$CLASS.h" \n
$+featuredefn \n
$classdefn \n""
```

4.1 Class structure

Every OPN class or transition maps into one or more C++ or Java classes. Each class requires functionality for creating, cloning and destroying instances of the class. Originally, in translating OPNs to C++, class constructors and destructors were used for creating and destroying instances. However, the fact that C++ constructors and destructors cannot make use of virtual functions as virtual functions (since the virtual function table is only established by the constructor and dismantled by the destructor), the above obvious approach does not work.

So, in order to provide a similar translation for both C++ and Java targets, we have defined our own standard class interface — a static create function for generating an instance of the
class, a clone function for duplicating an instance, and a destroy function for discarding the instance.

In addition to the above, all generated classes have some common features — they store the (textual) name of the instance, the (textual) type name of the class, the (simulated) time of instance creation, together with functions to examine these data items. For example, a boolean function \texttt{oftype(char* typename)} is provided which tests whether the current instance is of the type specified, or is a subtype of the type specified.

4.2 Overriding class components

As noted above, every OPN class or transition is mapped into one or more C++ or Java classes. Further, each component of an OPN class maps into data and function members of these classes.

OPNs allow for the flexible overriding of class components — a component of a subclass may override a similarly-named component in the superclass provided it is of the same kind (e.g. field, function, transition) and provided it is of compatible type. This is the sort of overriding allowed in the Eiffel language [17]. Unfortunately, C++ and Java only allow for the overriding of functions [2, 3]. (It is possible to redefine data members, with the new definitions serving to hide the parent definitions, but then there are two data members and there is no notion of dynamic binding as there is for virtual functions.)

The Java code generated for an OPN field therefore has a data member declaration in the parent class, but not in the child class. Thus only one data member is defined and coercion is used to ensure that the appropriate type value is available. In addition, a (virtual) function member is defined which is used to initialise, test, and reset the field. In the subclass, this function member is overridden.

In other words, the absence of virtual data members in Java (and C++) is circumvented by the use of virtual functions.

4.3 Notification management

The execution of nets is defined by the determination and firing of enabled steps, which consist of one or more transition modes. The process of determining the steps is not specified. One simple approach would be to repeatedly evaluate each transition to see whether it has an enabled mode. Clearly this is grossly inefficient since the enabled status of a transition is not modified until the tokens in the adjacent places are modified. Therefore, in implementing OPNs, we have adopted discrete event simulation techniques based on a time-ordered queue of (possible) pending events. If the firing of some step causes a change to part of the net, then objects or net components dependent on that change are notified, with the possibility that they choose to schedule further pending events. This approach is likely to provide significant efficiency gains particularly for loosely coupled nets.

In order to cater for this approach, each generated class has a test and a commit function. The commit function is used to commit an object to some change and notify dependent objects, while the test function is used by a dependent object to respond to such a change notification. The test function may respond in a variety of ways, depending on the object for which it is defined — a field or state component would reevaluate its integrity constraint to ensure that it is still satisfied; a transition would decide whether to schedule a pending event since it may now be enabled.
4.4 Reference management

The above subsections have considered the general functionality of the classes generated from an OPN specification. OPNs do not support an arbitrary reference semantics since some forms of Petri Net analysis (particularly invariant analysis) depend on knowing when objects (in this case tokens) are discarded. Therefore, it is not sufficient to know that some reference counting scheme has decreased the number of references to an object and that it may now be garbage collected. Instead, OPNs specify that tokens are self-contained — when a token is removed from a place, it can be discarded; and when a token is added to a place, it is guaranteed to be a new object. The semantics of OPNs thus specify when objects are created and destroyed.

In implementing OPNs in C++, it was necessary to determine when objects were discarded, so that the allocated memory could be reclaimed. For Java, the built-in garbage collection made this unnecessary. However, it was still important to maintain the appropriate notification mechanisms discussed in §4.3.

It turned out that both of these issues could be addressed in the generated code by a modified form of reference counting scheme. Instead of storing with each object the number of references to it, each object stores references to dependent objects (as required by the notification scheme of §4.3). If an object finds that it has no dependent objects, i.e. no other objects care about what happens to this one, then the object destroys itself.

Clearly, this reference counting scheme is unnecessary for memory management in Java (though it can be used to encourage early garbage collection). In any case, the underlying memory management is not visible to the OPN user.

4.5 Run-time support

There are a number of classes defined which supply generic properties and common run-time functionality for different kinds of OPN objects. We briefly consider each in turn.

The class Generic supplies the functionality common to all objects, as described in §4.2–4.4.

The class Token supplies the functionality appropriate to tokens in places. This includes the token’s status (i.e. selected or unselected for input, and proposed for output or resident), and functions which can evaluate the position of the token relative to others in the place, i.e. first, last, or delayed for a given period of (simulated) time. Given the duality between firing modes of transitions and tokens in places, the class Token is also used for transition firing modes.

The class Place supplies the functionality for places. This includes a list of the resident tokens and statistics about the tokens which are or have been resident at the place. Functions are provided to manipulate the list of tokens — get to select a token for removal, put to propose a token for addition, see to select a token for viewing, and add to deposit a token directly (as for example, in the initialisation of a place). In building OPN superplaces, it is possible to override these functions by transitions (see §2.3).

The class Trans supplies the functionality for transitions. The duality between places and transitions means that this class inherits from Place but redefines the functions test and commit so as to generate an enabled firing mode, and to fire the enabled modes.

In addition to the above classes, the run-time system includes a discrete event scheduler which drives the net. It maintains the list of pending events which is ordered by time (for efficiency reasons). However, for each time the events are randomly ordered, thus providing a certain level of non-determinism in the choice of the next event. The scheduler also tests transition modes for enabling, collects steps, and fires them. It is simpler to have a single global event scheduler, but the net semantics equally allows each transition to have its own independent scheduler. Any other level of coverage of a number of transitions by a scheduler is
possible. Elsewhere, we have considered the possible use of reflection to modify the scheduler to implement different simulation strategies [13].

5 Assessment of Java and C++ as translation targets

The above software development has led to some interesting comparisons of Java and C++ in terms of their suitability as a target for the translation process. In this section, we summarise our experiences, particularly in the light of the claimed advantages of Java over C++ [4].

5.1 Overriding data members

Given that OPNs allow the overriding of fields, the lack of support for this in both Java and C++ creates implementation difficulties which had to be resolved as in §4.2.

5.2 Constructors and destructors

Given the approach (discussed in §4.2) of implementing the overriding of fields using data members, coercion, and virtual functions (for initialising, testing, and resetting them), the initialisation of data members cannot occur in C++ constructors (see §4.1). Experience shows that this is not unique to this translation context, and consequently we believe that this severely limits the use of constructors in C++. By contrast, this is not a problem for Java where all functions are virtual and can be called in the constructor and destructor. Consequently, it is difficult in C++ to adopt the approach to robustness advocated by Eiffel (and adopted in OPNs), where the class invariant is required to hold following the initialisation of an instance, i.e. after the completion of execution of the constructor [17].

5.3 Memory management and garbage collection

The use of garbage collection in Java is touted as a significant advantage, since it removes a common source of errors [4]. In the context of this project, we agree with this assessment — it is not desirable for the user to be compelled to worry about allocation and deallocation of memory. Thus, memory management is not explicitly controlled by the user in OPNs. However, in our translation context, the semantics of OPNs dictate when memory will be deallocated and it would be desirable to be able to notify the garbage collector that memory is no longer required, possibly leading to more optimum allocation and deallocation.

5.4 Overloaded functions and default parameters

Both Java and C++ support overloaded functions. While not essential, this is used in the translation from OPNs for defining two clone functions for each class (see §4.1) — one which takes a minimal set of parameters (for cloning an object without any changes), and one which takes additional parameters (for cloning an object with specified modifications to exported components). Overloaded functions make it possible to use only one function name and therefore reduce the list of reserved identifiers which the user needs to be aware of and avoid.

In C++, we attempted to achieve the same result by having a single definition of the clone function, with the additional parameters being assigned default values. The interesting thing is that such a definition with additional optional parameters does not hide the version without additional parameters in the parent class. This means that it is necessary to define two versions of the clone function.
As in §5.2, this seems to be an awkward problem for C++ — while the language design choice seems reasonable, it leads to unexpected and obscure behaviour.

6 Extensions

In previous sections, we have considered how OPN specifications can be mapped into Java, thus providing animation and prototyping capabilities for such specifications. To a large extent we have been concerned with the suitability of Java as a target for this translation process.

In this section, we briefly consider the other side of the coin, i.e. the extent to which the Petri Net paradigm is suitable for harnessing the extended capabilities of Java. As noted in §1, there are numerous extensions to Java and this raises the question whether this functionality can be effectively utilised. One way of achieving this is to have a consistent, unifying paradigm.

6.1 Multithreading

For the purposes of this discussion, we classify multithreading as a Java extension even though it was supported in the original Java language [4]. Multithreading was provided so that applets, in the context of web browsers, might provide constant availability of services and might support multiple independent concurrent activities, such as loading web pages, playing audio clips, displaying graphics, etc.

The essence of multithreading in Java is the provision of separate threads of control and their synchronisation using locks associated with particular objects. Where a class declares methods as synchronized, the instances of the class act as a monitor, i.e. only one thread can be executing the methods of the class at any time.

As already noted, concurrency is fundamental to Petri Net theory. The firing rules allow transitions to fire at any time, thus supporting the notion of availability of services and concurrent independent processes. Further, the firing rules determine how multiple transitions can compete over the tokens in common input places. It should therefore be a simple matter to harness the multithreading capabilities of Java without modifying the semantics of a particular net.

6.2 Remote Method Invocation and Object Serialization

Remote method invocation and object serialization belong together.

"RMI enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses Object Serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism." [20]

"Object Serialization extends the core Java Input/Output classes with support for objects. Object Serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI). The default encoding of objects protects private and transient data, and supports the evolution of the classes. A class may implement its own external encoding and is then solely responsible for the external format" [20]

In other words, remote method invocation allows the components of a system to be distributed
across different machines, and object serialisation provides the mechanisms for transferring objects between those machines.

Again, we claim that the OPN formalism is ideal for harnessing these Java enhancements. It is inherent in nets that the communication between components is by token passing. In OPNs, tokens can be arbitrary objects but they are assumed to be self-contained. If parts of the net are allocated to separate machines, then the components can be initialised by the appropriate binding calls of RMI, and the function calls for token transmission become RMI calls. In addition, the transfer of a token between two parts of the net resident on different machines using object serialization is ideal, since it is known not to be referenced by other objects.

6.3 Java Beans

"JavaBeans extend Java's ‘write once — run everywhere’ capability to reusable component development. In fact, JavaBeans takes interoperability a major step forward — your code runs on every OS and also within any application environment. A Beans developer secures a future in the emerging network software market without losing customers that use proprietary platforms, because JavaBeans interoperate with ActiveX, OpenDoc, and LiveConnect." [20]

Java Beans provides a simple approach to building applets out of reusable components. The interaction between beans is by the transfer of events, which are a particular class (or classes) of objects. When beans are combined, each bean indicates the events it wishes to receive from other objects.

Again, this communication style is particularly appropriate to the Petri Net formalism with its interaction by token passing. There are striking parallels on the one hand between the distribution of events by a bean and the distribution of tokens by a transition, and on the other hand between the reception of events by a bean and the reception of tokens by a place. Thus, a bean has a natural parallel with a simple subnet as in fig 6.1.

![Fig 6.1: Subnet corresponding to a bean](image)

The tokens deposited in the place would be events which the bean needs to receive, and the tokens distributed by the transition are events sent to dependent objects. The connection of such beans by drawing arcs to indicate the event flow is the natural parallel of connecting Petri Net components. That being the case, it would be a simple matter to integrate pre-written beans into an applet produced using our proposed methodology. It would also be possible to use our methodology to produce bean components.

7 Conclusions and Further Work

OPNs provide great descriptive power for the modelling of concurrent systems. The use of a Petri Net formalism provides a useful graphical notation and allows us to build on Petri Net analysis techniques. Object-orientation provides powerful composition mechanisms for specifying complex systems.

This paper has proposed the use of OPNs as part of a software development process for producing Java applets for animating formal specifications. The paper has concentrated on the translation of OPNs into Java code. As a result of this exercise, it has presented a number of observations about the use of C++ and Java as targets for such a translation. It has also considered how some extensions to Java such as multithreading, distribution and persistence can be effectively harnessed in this unifying framework, without compromising the net semantics.
References


