Applications of Join Java

G Stewart Itzstein **David Kearney**

School of Computer and Information Systems
University of South Australia,
Mawson Lakes Campus, Mawson Lakes
Adelaide, South Australia 5095,
Email: itzstein@cs.unisa.edu.au

Abstract

We have previously proposed Join Java, a super-set of Java that incorporates the synchronisation and concurrency semantics of the Join calculus. Join Java incorporates asynchronous method calls and message passing. These modifications support the object-oriented flavour of Java. Synchronisation is expressed by a conjunction of method calls that will execute associated code only when all parts of the condition are satisfied. Thread creation is expressed by addition of a new return type that indicates an asynchronous method. This paper illustrates how Join Java can be used to represent two popular process semantics in a straight forward manner. In this paper we first examine state charts and how they map directly into Join Java syntax. We then examine Petri nets and how they also map into Join Java.

Keywords: Join Java, object oriented, state diagrams, state charts, Petri nets, programming, process calculi.

1 Introduction

Java has made concurrent programming using threads widely available to mainstream programmers. However, this situation has just re-emphasised what many experienced concurrent programmers already knew, that concurrent programming is inherently difficult. It is easy to make a mistake in a complex application that uses low-level synchronisation constructs such as monitors. Object oriented designs don't necessarily make concurrent programming easier. A poorly designed concurrent object oriented program can easily obscure the behaviour of threads running in parallel. Unlike processes in operating systems, which are protected by memory management software (other than those explicitly given all privileges), Java uses a type system to protect users from executing unsafe operations. However, the Java type system does not protect the user from concurrent access to shared variables. For example, programming a thread pool using only monitors can be a non-trivial task. The programmer needs to pass jobs (usually objects implementing the runnable interface) to a job dispatcher. This job dispatcher via some central registry of workers finds which threads are idle and signals a thread that a job is waiting. The worker thread then collects the job and runs it returning the outcome via some shared variable. Implementations of this pattern can be quite difficult to get right with shared data being vulnerable to double updates.

With the increasing interest in concurrent applications such as enterprise information systems, distributed programs and parallel computation there seems to be a need to provide a higher-level abstraction for concurrency synchronisation. This will provide the ability to directly represent much higher-level abstractions for concurrent programming.

The Join Java compiler generates standard bytecode allowing compiled code to run on any Java platform. Using synchronisation in Join Java is as straightforward as writing and calling standard Java methods. We have previously shown (Itzstein & Kearney 2001) how Join Java incorporates concurrency semantics from the formal Join calculus (Fournet, Gonthier, Lvy, Maranget & Rmy 1996) and that this would allow a more rigourous investigation of the behaviour of implementations, potentially reducing subtle errors.

It could be argued that many of the advantages claimed above could be achieved by using a pre compiled library of high-level concurrency classes. In fact a number of approaches have been taken in this direction (Hilderink, Bakkers & Broenink 2000, Welch 1998), a lot of which follow the approach of Hoares (1980) Communicating Sequential Processes. However extending the language rather than supplying a library allows the compiler to better utilise the resources used to support concurrency, as the synchronisation mechanism is an integral part of the language. For example, mandatory use of libraries are difficult to enforce, a programmer may choose to use one feature but not another (or even forget to call a method at the correct time) which leads to potential

Copyright ©2002, Australian Computer Society, Inc. This paper appeared at the Seventh Asia-Pacific Computer Systems Architecture Conference (ACSAC'2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 6. Feipei Lai and John Morris, Eds. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

problems at runtime. If the language is implemented as a pre-processor any syntax errors are related to the programmer in terms of the output of the pre-processor not the source file that the programmer is familiar with. In this paper when we refer to Java we mean the language itself and not necessarily the many libraries (such as the API (Gosling & McGilton 1995) or Triveni (Colby, Jagadeesan, Jagadeesan, Laufer & Puchol 1998)) that have been added to support particular application domains.

The paper begins by providing a motivation for the language extension. Section three and four provides a brief overview of the Join Java extension. In section five we describe state charts and how they can be modelled in Join Java. Section six describes Petri nets and how they too can be modelled. Section seven draws conclusions and suggests some refinements. Section eight looks, at current and future work.

2 Motivation

Thread programming in Java is not easy. More and more programs written in Java are multi threaded in someway (viz. Distributed programs, Enterprise information servers etc). Consequently it is necessary to always keep in mind that the code could be accessed in a multi threaded context. A solution is to synchronise all code within a program so that object safety is guaranteed. However, Holub (2000) shows that this approach greatly slows down the execution of the program. Consequently, programmers writing Java code have to be proficient in the low level threading mechanisms of Java. It is then necessary to provide a mechanism where programmers can easily pass messages between threads of execution without worrying about synchronising shared variables.

3 Join Calculus

In this section we give an overview of the Join calculus (Fournet et al. 1996) and introduce some of the terminology we have adopted. Join calculus can be regarded as a functional language with Join patterns. The Join Java extension semantics are based on the Join algebra originally proposed by Fournet. This calculus can be thought of as both a name passing calculus (i.e. processes and channels have identifiers) and a core language for concurrent and distributed programming (Maranget, Fessant & Conchon 1998). Traditionally Join operational semantics are specified as a reflexive chemical abstract machine (CHAM) (Berry & Boudol 1992, Maranget et al. 1998). Using this semantic the state of the system is represented as a "chemical soup" that consists of active definitions and running processes (Maranget et al. 1998). Potential reactions are defined by a set of reduction rules. When the soup contains all the terms on the left hand side of a reduction rule the terms react and generate all the terms on the right hand side of the reduction rule. This is shown in figure 1 where the addition of term **B** creates new terms (via the reduction rule) **D** and **C** on the right (shown with stars in the figure) and removes both terms **A** and **B**.

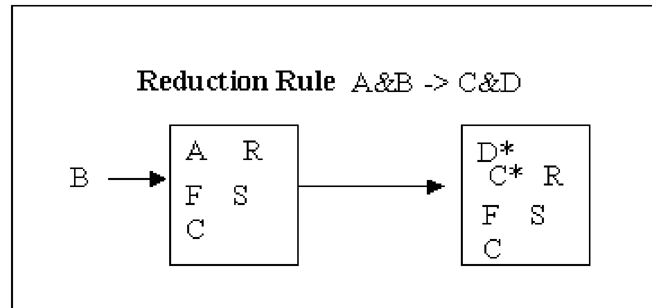

Figure 1: Operational Semantics of the Join Calculus

In Join Java we call the individual terms on the left hand side of the reduction rule, *Join fragments*. We call the entire left-hand side of the reduction a *Join pattern* and the entire rule a *Join method* When all the fragments required to fulfil a Join pattern exist in the soup (in our case a pattern matcher) the body of the Join method is executed.

Join patterns can also be viewed as guards on the message passing channel. When all the fragments of the Join pattern are called the "guarded" message is transferred between callers of the Join fragments. The standard Join calculus does not support synchronous Join fragments however, a formal translation based on CPS (Steele 1978, Appel 1992) is available from (Fournet et al. 1996, Maranget et al. 1998) INRIA to convert expressions containing synchronous names to asynchronous fragments. Join calculus patterns can thus be mapped directly to Join Java patterns. For a formal definition of the Join calculus semantics consult Fournet *et al.* (1996).

4 Join Java

In this section we will introduce the syntax of our super-set of Java. Join Java makes two syntactic additions to Java.

1. Addition of Join patterns.
2. Addition of a signal return type.

4.1 Addition of Join Patterns

A Join method (see figure 2) in Join Java gives the guarded process semantics of the Join calculus (Fournet et al. 1996, Odersky, Zenger, Chen & Zenger 1999) to Java. That is the body of a Join method will not be executed until all the fragments

of a Join pattern are called. The leading Join fragment of a Join pattern can be viewed as a normal method in Java. If Join patterns are defined with pure Java return types such as **void** or **int** they have blocking semantics. If the return type of the leading fragment is the new type **signal** the method is asynchronous (an early return type). Trailing Join fragments are always asynchronous in the current version of the language, that is they will not block the caller. A non-Join Java aware class can call methods in a Join Java class even if the return type is **signal**. In the case of a **signal** return type the caller will return immediately. In figure 2 we can see a example of a Join method declaration.

```
final class SimpleJoinPattern {
   int A() & B() &C(int x) {
      //will return value of x
      //to caller of a
      return x;
   }
}
```

Figure 2: Join Method Declaration

Figure 2 method would be executed when calls are made to all three methods (*A() B()* and *C(int)*). A call to method *A()* will block the caller at the method call until methods *B()* and *C(int)* are called due to the requirement that a value be returned of type *int*. When all methods have been called the body of the method is executed returning the *int* value to the caller of *A()* The message passing channel in the example is therefore from the caller of *C()* to the caller of *A()* as there is an integer value passed from the argument of *C()* to the return type of *A()*. The call to *B()* only acts as a condition on the timing of the message passing.

An interesting treatment of the issues involved in the interaction of inheritance with a simple object oriented language is available from (Fournet, Laneve, Maranget & Remy 2000). In the current version of our language extension all classes that make use of Join patterns must be declared as final. This neatly avoids complications arising from inheritance complications in Fournets paper.

The final change to Java related to the introduction of Join patterns is the addition of two class modifiers **ordered** and **unordered**. These modifiers alter the behaviour of the pattern matcher for the Join patterns of the current class. Firstly, **unordered** (the default behaviour for a Join Java class) exhibits non-deterministic behaviour when confronted with multiple possible matches. For example figure 3 shows a Join Java class in which we have two transitions and a constructor. When a object of type UnorderedExample is created method *A()*, *B()* are called followed by *S()*. When *S()* is called we have a non-deterministic choice to make. Both transitions one and two can be matched but not both. With the **unordered** modifier the pattern matcher will make a random determination which method to complete. However if the modifier **ordered** is used the pattern matcher will give precedence to the first pattern that is defined in the class (hence the designation **ordered**), in this case transition one.

```
final class UnorderedExample {
   //Constructor
   UnorderedExample() {
      B();
      A();
      S();
   }
   signal A() & S() {
      System.out.println("Transition1");
   }
   signal B() & S() {
      System.out.println("Transition2");
   }
}
```

Figure 3: Join Method Declaration

4.2 Asynchronous Return Type

A single Join fragment with a **signal** return type indicates that the method is asynchronous. Any method with a **signal** return type specifies that on being called a thread will be created and started. Figure 4 shows an example declaration of a thread with argument x.

```
final class SimpleJoinThread {
   signal athread(int x) {
      /*code that uses x*/
   }
}
```

Figure 4: Join Method Declaration

4.3 Concurrent Hello World

By combining together the techniques illustrated in figures 2 and 4 we can build a concurrent hello world program. Figure 5 shows a program that has two thread definitions *athread1* and *athread2*, the first thread passes the phrase "helloworld" to the second thread via a Join method which prints the message. We can see that this is a simple unidirectional communication channel between two threads *athread1* and *athread2*.

The syntax of the language extension is provided in figure 6. For more complete coverage of Join Java see Itzstein (2001)

5 State Diagrams and Charts

This section will show how both state diagrams and charts can be mapped into the Join Java language.

```
final class HelloConcurrency {
   public HelloConcurrency() {
      //constructor
      athread1();
      athread2();
   }
   signal athread1() {
      //thread sends com to
      //other thread
      channelIn("HelloWorld");
   }
   signal athread2() {
      //thread receives com from
      //other thread
      System.out.println(channelOut());
   }
   String channelOut()&channelIn(String val) {
      //we could write code here
      //to be executed when the pattern is
      //matched and a channel is opened.
      //however in this case we just send
      //the data
      return val;
   }
}
```

Figure 5: Message Passing Hello World

```
JoinMethod = [Modifier] Return-type IdentFrags{stms}
Modifier   = public | private | protected
Return-Type= signal | UserDefined | BaseType
IdentFrag  = Ident(ParamList)[&IdentFrags]
Ident      = any legal Java identifier
ParamList  = standard Java method parameters
Stms       = standard Java method body
UserDefined= standard Java class types
BaseType   = standard Java base types

A Join method can be placed at any point that
a normal Java method can be legally placed.
```

Figure 6: Join Java Language Extension Syntax

5.1 State Diagrams

State diagrams (and charts) represent a system as a series of states (circles) connected by transitions (arcs connecting states). A transition usually indicates the condition in which the system will change from a given state to another state connected by the transition. Being graphs the most common representation is an adjacency matrix. These adjacency matrices are interrogated when an event occurs and a shared variable indicating the current state is updated. Applications that use state representation usually have the transition introduce data into the system. For example, in a state machine representing an automatic teller the transition *deposit-money* would introduce the amount into the depositing state. This leads us to a disadvantage of the normal adjacency matrix representation in that transitions have no mechanism to introduce data directly into the destination state. To do this we have to provide additional data or control structures to support these values.

Join Java makes the conversion of state charts to an object oriented syntax representation straightforward. The Join Java representation uses Mealy machine (1954) interpretation where executing code in the body is associated with a state transitions rather than the state itself. When coding a Join method to represent a transition the Join fragments represent first the transition call, followed by the state the machine must be in. Within the Join method a call to the destination state is made to indicate the new state. To illustrate, using the diagram from figure 8, figure 7 shows a method representing the transition from state **A** to state **B**.

```
final class StateDiagram {
   void x() & A() {
      C();
   }
}
```

Figure 7: State Transition

We could read this as transition **x** can be completed when and only when the machine is in state **A**. When the method is executed the Join fragment (we could also call this a token) **A** representing the state is consumed and a new Join fragment **C** is created, that is the machine is now in state **C**.

Branching is carried out by introducing a rule for each transition out of a state. Using figure 8 as an example again. We can add to the transition we mentioned previously a second transition from **A** to **B**. We form the appropriate rule *b()* & *A()* { *B()* }. Now if we have the situation where the system is in state **A**. The decision which Join method to call (and hence what state to transition to) is made at run time by the occurrence of the call (transition) to either *x()* or *b()*. In the event of two calls being made at the same time the system selects the first call that is given priority by the pattern matcher. It would initially appear that the system is opportunistically checking for pattern matches however, this is not the case, as the system only checks for matches when the pattern matcher is notified of a change in the waiting fragments. In other words the runtime system is reactively checking for matches only when the state of the pool changes. Determinism is hard to implement on a standard Java virtual machine as they vary between implementations on different architectures (vis Windows vs Solaris implementations of threads). Consequently to be consistent with the underlying system Join Java allows the virtual machine to make the decisions about priority of thread execution as per its own implementation.

Figure 8 shows a state diagram with states **A**, **B** and **C** and transitions **a**, **b**, **x** and **y**. We only need to specify one Join pattern per transition (or edge) and an error method to absorb any transition calls made when the machine is not in the correct state. The completed code that represents the full state diagram in figure 8 is given in figure 9. We can see that in the diagram below that there is a direct mapping between transitions and methods. We also need to set the

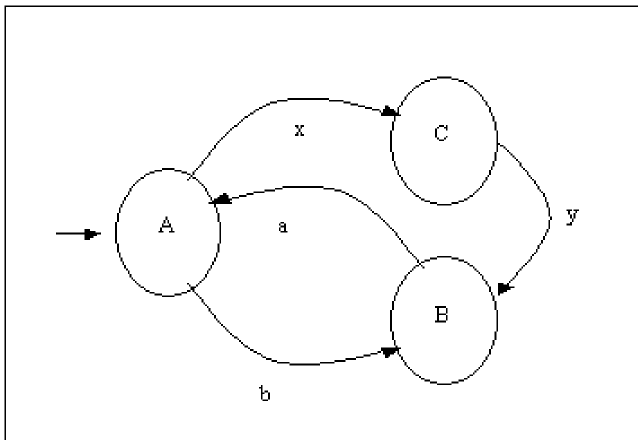

Figure 8: State Diagram

initial state (indicated by the arrow on the left of state **A**) and hence the starting state of the machine. This is done by calling *A()* in the constructor. Also note the usage of the **ordered** class modifier, this is used to make sure the transition methods are fired first if possible in preference to the error methods at the end of the class.

```
\\note the ordered modifier. this means
\\match the first pattern that is defined
\\in preference
final ordered class StateDiagram {
   //constructor
   StateDiagram() {
      //constructor sets initial state
      A();
   }
   void x() & A() {
      //code to execute on transition x
      C();  //call to destination state
   }
   void b() & A() {
      //code to execute on transition b
      B();  //call to destination state
   }
   void a() & B() {
      //code to execute on transition a
      A();  //call to destination state
   }
   void y() & C() {
      //code to execute on transition y
      B();  //call to destination state
   }
   //if one of the patterns above doesn't
   //match these will match.
   //this removes calls that get called
   //when the machine is not in the correct
   //state
   void x() {}
   void b() {}
   void a() {}
   void y() {}
}
```

Figure 9: State Diagram Join Java Code

5.2 State Charts

Similarly, concurrent state charts can be converted by adding the inter-chart transformations as additional Join fragments in each transition. Figure 10 gives a state chart in which we can describe via the Join Java code in figure 11.

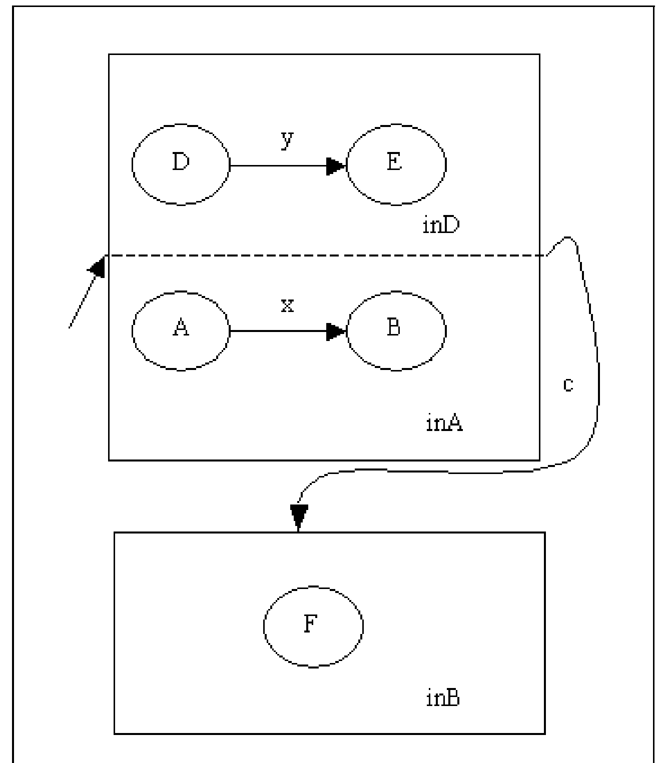

Figure 10: State Chart

Again the initial state is represented by calls in the constructor. In state charts, as we have an extra level of abstraction, we also have to provide extra Join fragments showing which chart contains active states. Using our example in figure 10 we see that we have a transition between **D** and **E** called **y**. This transition resides within chart **inD**. Therefore the condition for transition **y** to be called is that we are in state **D** and in chart **inD**. As a Join pattern it is *y()* & *A()* & *inD()*. The important point in this example is that when a transition occurs from one chart to another we need to use up any state information left in the state space. In the case of transition **c** (**inD** to **inB**) we have the **inD** token as well as the possibility of a **D** or an **E** state. We then have to clean the pool of unused state information. We do this by creating Join Patterns that will consume anything that is left in the event that we leave a state chart. In our **inD** chart we need to consume both the **D** and **E** states. This is illustrated in figure 11 below where *leaveD()*&*D()* and *leaveD()*&*E()* Join patterns dispose of any state information that is left. Figure 11 shows the complete code for the state chart in figure 10. A more object oriented approach of abstracting the charts is to build a new class for each individual chart. We then throw away the object when we move to another chart. This removes the necessity of cleaning the pool as we leave the chart.

```
final class StateChart {
   //constructor
   StateDiagram() {
      //constructor sets initial state
      D();
      A();
      inD();
      inA();
   }
   void y() & D() & inD(){
      //code to execute on transition y
      E();     //call to destination state
      inD();   //make sure it is still in
               //sub-state inD
   }
   void x() & A() & inA(){
      //code to execute on transition x
      B();     //call to destination state
      inA();   //make sure it is still in
               //sub-state inA
   }
   void c() & inA() & inB() {
      //create a new state chart
      inBClass ds = new inBClass();
      //code to execute on transition a
      ds.F();  //call to destination state
      inB();   //make sure it is still in
               //sub-state inA
      LeaveA();
      LeaveB();
   }
   //clean up/take unused state
   //info out of pool
   void LeaveA() & A() {}
   void LeaveA() & B() {}
   void LeaveB() & D() {}
   void LeaveB() & E() {}
}
```

Figure 11: State Transition Join Java Code

6 Petri nets

Petri nets (Petri 1962) are similar to state charts in that they allow the dynamics of a system to be modelled graphically. In addition the Petri nets can also represent the system mathematically allowing more rigourous examination of system properties.

A Petri net is composed of four structures, a set of places, a set of transitions, a set of input functions and a set of output functions (relative to the transition). The structure and hence the behaviour of the modelled system is defined by the configuration of places, input functions, output functions and transitions. Places can contain a token that will allow a transition to occur. The presence of a token indicates that a transition may be available, if and only if there are tokens waiting on all other inputs to the transition. A detailed coverage of Petri net theory can be found in (Peterson 1981).

6.1 Structures

Diagrammatically Petri nets have circles indicating places, blocks indicating transitions and arcs between places and transitions indicating input and output functions. Figure 12 illustrates the different symbols of a Petri net.

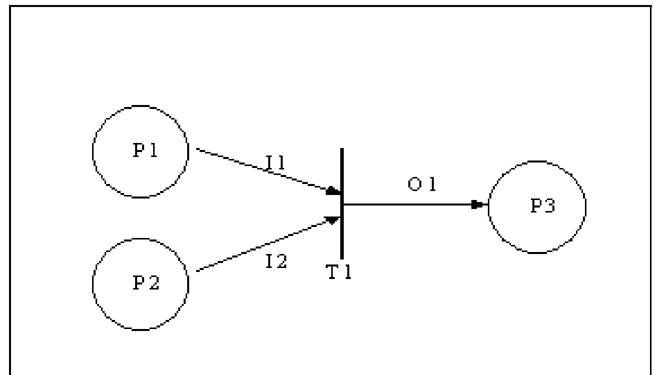

Figure 12: Simple Petri Net

We can see in figure 12 we have three places named **P1**, **P2** and **P3**. Two input functions **I1** and **I2**, A transition **T1**, and a single output function **O1**. In this case if there are tokens waiting in both **P1** and **P2** then a transition **T1** will be executed and a single token will be placed in **P3** removing a token from both **P1** and **P2**.

As in the state chart implementations there exists a fairly easy and mechanical structure for converting a Petri net implementation into a Join Java program. As Petri nets are transition centred the translation is almost direct. In the examples presented we will assume that the transitions fire immediately they are enabled.

In the next two sections will cover two different types of Petri nets. Firstly unbounded Petri nets, these nets allow multiple tokens to exist in the places at any one time. The second type covered will be one-bounded place Petri nets, these networks only allow one token per place in the network.

6.2 Unbounded Petri nets

Unbounded Petri nets whilst being the more complex are the easiest to implement in Join Java. One needs to produce a method for each transition. The Join method is composed of the places tokens must exist (preconditions). Each Join method will only be fired when at least one token exists in each of the input places. The body of the Join method contains the post conditions such as the destination place as well as any side effects that the transition may entail.

6.2.1 An Example in Join Java

Using figure 13 as an example we can see that there are tokens available on both places **P1** and **P2**. This allows transition **T1** to fire (calls to both **P1** and **P2** are available in the pool). The method will consume **P1** and **P2** tokens and create a **P3** token. Figure 14 shows the state of the Petri net after the transition.

The code for this Petri net (figure 15) shows how one would write the code to represent this net.

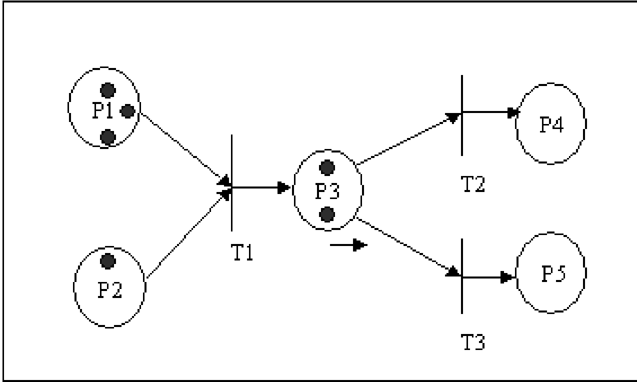

Figure 13: Non Bounded Petri Net Before Transition

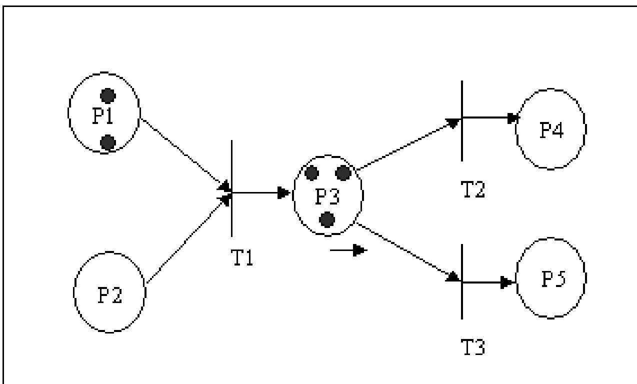

Figure 14: Non Bounded Petri Net After Transition

```
final class PetriNet {
   //transition 1
   signal P1() & P2() {
      P3(); //P3 has now got a token
   }

   signal P3() {
      P4(); //P4 has now got a token
      P5(); //P5 has now got a token
   }

   signal P4() {
      //No transitions out of this
   }

   signal P5() {
      //No transitions out of this
   }
}
```

Figure 15: Non Bounded Petri Net Join Java Code

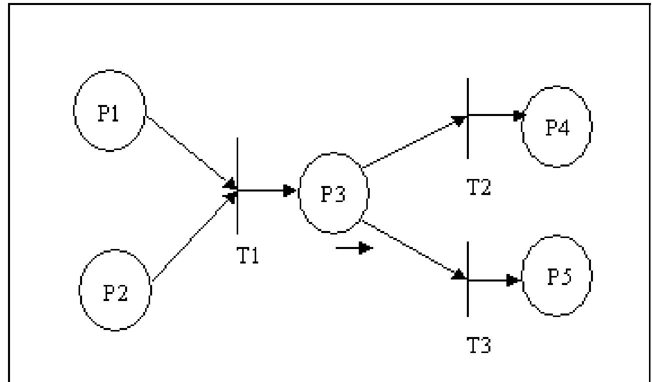

Figure 16: Bounded Petri Net Example

6.3 Bounded Petri nets

One-bounded Petri nets are similar to unbounded Petri nets except that any one place can only have one token. This means in a one-bounded Petri net a transition will not fire if there is a token already waiting in the output place of the transition.

6.3.1 An Example

We take the example of a one bounded buffer Petri net based on the unbounded Petri net in figure 13. In a bounded Petri net that there is the requirement that the destination place not have a token residing in it when the transition is triggered. In our implementation we handle this by having a "not" token indicating the lack of a token in the destination place. In other words a place will have either a token (which shares the name of the place) or a not-token.

Translation of figure 16 is a matter of taking each transition and place and converting them into a method. Using transition **T1** of the figure as an example we define the places that must have tokens; in this case **P1** and **P2**. We then specify that **P3** must be empty (i.e. a not-token). Therefore the condition on which **T1** will be activated is if *P1()* & *P2()* & *notP3()*. Once the condition for the execution of the **T1** transition is defined we now specify the post condition for the transition. In this case **P3** having a token and **P1** and **P2** being empty. That is *notP1()*, *notP2()* and *P3()*. Figure 17 shows the code for the entire Petri net. Note that in addition to the transitions we also need to initialise the network. We do this via the constructor which initialises the Petri net to the starting state. We also need to define any terminal places (places with no output transitions) as normal methods.

It should be noticed that in this example the return types of the methods are **signal**. The caller of the method does not wait for execution of the Join method to complete successfully. The other point to note is that the methods do not pass arguments in the example provided. By adding arguments and return types to the Join methods we can pass information between places (via the transitions). Conceivably this allows the application of the technique to more than just elementary Petri nets. There are a number of non-elementary style Petri nets such as coloured Petri nets (Jensen 1986) that this language could support via arguments in the Join fragments. However,

```
final class PetriNet {
   //Constructor sets up initial state
   //of PetriNet
   //In this case tokens in places 1,2 and 5
   PetriNet() {
      P1();
      P2();
      notP3();
      notP4();
      P5();
   }

   //Transition 1
   signal P1() & P2() & notP3() {
      notP1();  //P1 is now empty
      notP2();  //P1 is now empty
      P3();     //P3 has now got a token
   }

   //Transition 2
   signal P3() & notP4() {
      notP3();  //P3 is now empty
      P4();     //P4 has now got a token
   }

   //Transition 3
   signal P3() & notP5() {
      notP3();  //P3 is now empty
      P5();     //P5 has now got a token
   }

   void P4() {
      //end of net
   }

   void P5() {
      //end of net
   }
}
```

Figure 17: Bounded Petri Net Join Java Code

in this paper we have restricted ourselves to covering only elementary Petri nets.

6.4 Partial Three Way Handshake Petri net

A more complex example of Petri nets in Join Java is provided in figure 18. This diagram shows a Petri-net illustrating the opening stages of a TCP three way handshake from the point of view of the client. In this example the system starts in the **CLOSED** place and attempts to transition to the **ESTABLISHED** place. The line that travels clock-wise around the diagram indicates the normal path of a client making a connection to a server. The **LISTEN** place is the point that the server would normally wait. For the sake of brevity we have restricted ourselves to the clients path. This diagram is more complicated as it has unnamed places indicated by light grey dots. We have added labels, **SSYN1**, **SSYN2**, **SYNACK1** and **RSYNACK1** in order to make coding of the Petri net in Join Java easier. If we were to omit these places we would have to combine the transitions on each side of the place into a single transition possibly changing the meaning of the net. The aim of this net is to step through the process of hand shaking between a client and a server on a TCP connection. The advantage of having a complicated handshaking protocol is that there is no chance of confusion between the client and server in respect to which state the other machine currently resides. In the event of a problem the client and server will return to the **CLOSED** place.

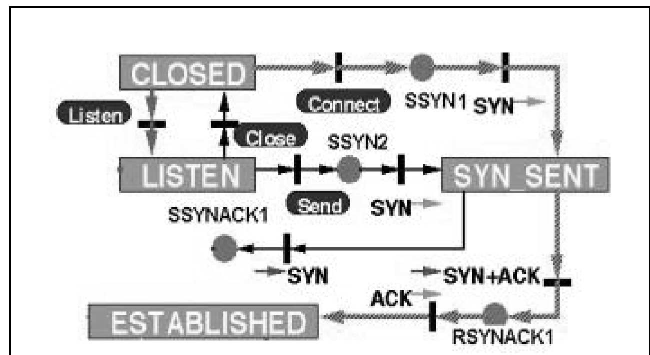

Figure 18: Partial Three Way Handshake

Using the first transition we can see that when **Connect** is enabled the token moves from **CLOSED** to **SSYN1**. To convert this into Join Java we state the requirements of the transition. That is that we are in place **CLOSED** and transition **Connect** is enabled, in Join Java this would be *Connect(Socket) & CLOSED()*. We then state the post conditions, in this case a token resides in place **SSYN1**, in Join Java this would be *SSYN1(Socket)*. As completed code the Join Java method would look like figure 19.

```
signal Connect(Socket x)&CLOSED() {
   SSYN1(x); //place SSYN1 has now got the token
}
```

Figure 19: Petri Net Join Java Method

Two points should be noted in this example. Firstly, the method *Connect(Socket)* passes arguments into the Join method. These arguments allow the new place to receive information about the current connection via arguments in the call to the Join fragment *SSYN1(Socket)*. This information would be useful in a full implementation, as the system would need to record socket information in order to communicate with the server. Secondly, the "not" token is not required in this implementation as we know that there will only ever be one token in the network and hence we do not need to worry about checking for tokens in the destination places.

Figure 20 shows the full class containing all the transitions required for a successful negotiation between a client and server. As usual we need to include the constructor that would initialise the Petri net. The method calls *SYN()*, *SYNACK()*, and *ACK()* in a real implementation would be more complicated as they would need to signal the success or failure of the packet that is sent or received from the server.

```
class Socket {}

final ordered class TCPHandShake {
   //These methods are called from outside
   //Connect(Socket);,SYNACK();

        //Constructor
   TCPHandShake() {CLOSED();}

   //Transition (1) from CLOSED place to
   //SSYN1 place.
   signal Connect(Socket x) & CLOSED() {
      //send syn to server
      //when complete call SYN()
      //to show that the message
      //has been sent
      /*send syn to server*/
      SYN();
      SSYN1(x); //SSYN1 place has now got
                //the token
   }

   //Transition (2) from SSYN1 place to
   //SYNSENT place.
   //The SYN() method in this transition
   //would be called by the network software
   //when the syn is received.
   signal SYN() & SSYN1(Socket x) {
      SYNSENT(x);
   }

   //Transition (3) from SYNSENT place to
   //RSYNACK1 place.
   signal SYNACK() & SYNSENT(Socket x) {
      //send ack to server
      //when complete call ACK()
      //to show that the message
      //has been sent
      /*send ack to server*/
      ACK();
      RSYNACK1(x);
   }

   //Transition (4) from RSYNACK1 place to
   //ESTABLISHED place.
   signal ACK() & RSYNACK1(Socket x) {
      ESTABLISHED(x);
   }

   //ESTABLISHED place
   signal ESTABLISHED(Socket x) {
      // client code to handle
      // application
      Close(x);
      // to start of close sequence
   }

   //commence close procedure
   void Close(Socket x) {}

   //Any other return to CLOSED place
   signal Connect(Socket x) {CLOSED();}
   signal SYN() {CLOSED();}
   signal SYNACK() {CLOSED();}
   signal ACK() {CLOSED();}
}
```

Figure 20: TCP Handshake Class

This example illustrates how we can set the conditions for progress through the network by adding methods that an external agent may use. In this case the *Connect(Socket)* method above would be called by some external agency (perhaps the user clicking on the connect button in a User Interface). This would allow the network to fire and transition to the second place in the network. Each place in the network would either send a message and move to the next place or wait for the receipt of a message from outside. We again used the **ordered** keyword as we want to handle the messages that arrive in the incorrect order by returning the token to the **CLOSED** place. In this case we send the token back to the start position by calling *CLOSED()*.

7 Conclusion

In this paper we have provided a brief overview of the Join Java language. We showed how that with two small additions to the syntax of Java we have added a powerful concurrency synchronisation mechanism to the language. Secondly we illustrated how state diagrams and state charts can be mapped into Join Java code. We then showed how the language could represent Petri nets allowing complex systems to be straight forwardly translated into object-oriented code.

Join Java better represents concurrency as it allows other formalisms to be directly expressed in the language with a minimum change from the core Java language. The structure of Join Java allows message passing to be easily implemented between different processes without having to concern oneself with the low level details of the message passing implementation.

We have shown that the Join Java language extension can directly map process semantics to an object oriented language. There are a number of advantages to having the language support the syntax directly.

1. This will help decrease development costs, as system design can be carried out using Petri nets. The Petri net design can then be translated directly to code. Maintenance is a lot easier as Petri net representation can be directly modified in the code.

2. The language itself carries out a lot of the work synchronising interactions and handling communications without making the programmer worry about low-level operations (the use of **wait** and **notify** are virtually unnecessary as they are completely handled by the runtime system of the language). This should reduce the occurrence of errors in the software increasing quality for little cost.

8 Current Work

At this time we have a simple prototype of the language based upon the extensible compiler developed by Matthias Zenger (2001). The prototype supports in addition to any standard Java program, both Join methods and asynchronous types. The language still needs rigourous testing and we are currently designing a set of regression tests in order to further test the stability of the language. We are currently extending the language so that the pattern matcher accepts object types and more particularly interfaces. The ability to pass an interface is useful as it opens up possibility of abstraction of synchronisation rules from the threads via interfaces. This should be completed at the same time as the pattern matcher is updated with a more efficient algorithm. We are also examining standard concurrency problems and how they translate to the Join Java language. A number of larger real world style applications will be built in the immediate future in order to benchmark the performance of the language.

Even though these extensions make programming easier to accomplish we still have to prove that the speed of the language extensions are sufficient for big applications. Whilst small trials have shown sufficient speeds we need to try larger applications and show that the system is scalable. Measuring the difference in two coding styles is exceedingly difficult as one needs to take into account not only the raw speed of execution and writing but also the quality and length of the code.

Alternate pattern matching algorithms will also be looked at to test the differences between different implementations of the language. At present the pattern matcher uses a tree structure to represent patterns. Optimisation is achieved by using levels of indices that require only subsets of the tree to be examined whenever there is a change inside the matching pool. However, further optimisation could be achieved by building trees that only represent nearly finished patterns. In this way only Join Patterns that are on the verge of completion will be examined further optimising the pattern matcher.

9 Acknowledgements

We would like to thank Professor Martin Odersky and Matthias Zenger from Ecole Polytechnique Federale de Lausanne for their assistance with this research.

References

Appel, A. W. (1992), *Compiling with Continuations*, Cambridge University Press, Cambridge, MA.

Berry, G. & Boudol, G. (1992), 'The chemical abstract machine', *Theoretical Computer Science* **1**(96), 217–248.

Colby, C., Jagadeesan, L., Jagadeesan, R., Laufer, K. & Puchol, C. (1998), Design and implementation of triveni: A processalgebraic api for threads + events, *in* 'International Conference on Computer Languages. 1998', IEEE Computer Press.

Fournet, C., Gonthier, G., Lvy, J., Maranget, L. & Rmy, D. (1996), 'A calculus of mobile agents.', *Lecture Notes in Computer Science* **1119**.

Fournet, C., Laneve, C., Maranget, L. & Remy, D. (2000), Inheritance in the join calculus, *in* S. Kapoor & S. Prasad, eds, 'FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science', Vol. 1974, Springer-Verlag, New Delhi India, pp. 397–408.

Gosling, J. & McGilton, H. (1995), 'The java language environment'.

Hilderink, G., Bakkers, A. & Broenink, J. (2000), A distributed real-time java system based on csp, *in* 'Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000', IEEE, Newport Beach, California, pp. 400–407.

Hoare, C. A. R. (1980), Communicating sequential processes, *in* R. M. McKeag & A. M. Macnaghten, eds, 'On the construction of programs – an advanced course', Cambridge University Press, pp. 229–254.

Holub, A. (2000), *Taming Java Threads*, 1st edn, Apress, Berkeley.

Itzstein, G. S. & Kearney, D. (2001), Join java: An alternative concurrency semantic for java, Technical Report ACRC-01-001, University of South Australia.

Jensen, K. (1986), 'Colored petri-nets', *Lecture Notes in Computer Science* **254**(Advances in Petri-nets), 248–299.

Maranget, L., Fessant, F. L. & Conchon, S. (1998), 'Jocaml manual'.

Mealy, G. H. (1954), 'A method of synthesizing sequential circuits', *Bell System Technical Journal* **34**(5), 1045 – 1079.

Odersky, M., Zenger, C., Chen, G. & Zenger, M. (1999), A functional view of join, Technical ACRC-99-016, University of South Australia.

Peterson, J. L. (1981), *Petri Net Theory and The Modeling of Systems*, Prentice Hall Inc, Englewood Cliffs N.J.

Petri, C. A. (1962), Kommunikation mit automaten, Phd, University of Bonn.

Steele, G. L. (1978), Rabbit: A Compiler for Scheme, Masters, MIT.

Welch, P. H. (1998), Java threads in the light of occam-csp, *in* P. H. W. Bakkers & A. W. P., eds, 'Architectures, Languages and Patterns for Parallel and Distributed Applications', Vol. 52 April http://www.cs.ukc.ac.uk/pubs/1998/702 of *Concurrent Systems Engineering Series*, WoTUG IOS Press, Amsterdam, pp. 259–284.

Zenger, M. & Odersky, M. (2001), Implementing extensible compilers, *in* 'ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages', Budapest.