Source to Java Translation: Issues and Experiences

Judith Bishop

Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa
Email: jbishop@cs.up.ac.za

Abstract.
Java is rapidly becoming the language of choice in a wide range of applications from networking to scientific to business. Other languages can gain the power of Java by compiling to Java or to its bytecode. Source to Java translation is an attractive option because it is faster to implement and offers advantages in retaining the structure of the original program as well as the possibility of linking in with Java classes which use graphics, networking and so on. Yet translating to Java presents certain challenges, and these can be overcome in a variety of ways. This paper surveys twelve projects which have used source to Java translation and evaluates the pros and cons of the approach. It then goes on to consider in detail a new source to Java translator for Darwin, and architectural description language. The target of the translation is not just Java, but also a communication system for distributed processing, which was also completely redesigned using Java's RMI. The system was successfully implemented and has no performance degradation compared to the previous C++ version.
Keywords: Java, Darwin, source translation, polymorphism, interfaces, RMI

1 Introduction

When Java first came out in the mid-nineties, there was a flurry of source-to-source translators built which enabled existing languages, both general and special-purpose, to piggy-back on Java's evident advantages. Examples of such languages across a wide range were Eiffel [Potter 1997] and Prolog [Banbara and Tamura 1999] as well as NESL [Hardwick and Sipelstein 1996] for parallel processing and JLAPACK for numeric libraries [Doolin *et al* 1998]. At the same time, the programming language community worked on taking Java where it had not been before by adding a selection of favourite features, such as polymorphism and higher-order functions, and implementing these through source translation once again [Krall 1997, Myers et al 1997, Odersky and Wadler 1997]. Others who had smaller languages to deal with, such as Basic [Lehman 1996], or access to the proprietary source of compilers, as in the case of Ada [Taft 1996], were more tempted to translate immediately to the Java Virtual Machine's bytecode. Those who translated from Smalltalk also envisaged using bytecode [Kawa by Bothner 1997, Bistro by Boyd 1999].

Five year's on, it would seem that the JVM is well-enough known, that the efficiency to be gained by translation to bytecode has made this approach a sensible choice. For a complete list see [Tolksdorf 2000]. Yet source-to-Java translation is still done, and was the choice of our project for the upgrade of Darwin, and architectural description language for configuring distributed programs, developed at Imperial College [Magee *et al* 1995].

The purpose of this paper is twofold: to summarise the advantages and issues presented by source-to-Java projects to date; and to describe and evaluate our recent experience with a Darwin to Java translator. This translation included taking into account communication via portals which were implemented by RMI on the Java end and embodied in the original runtime environment, Regis, which we rewrote as Jaden. Our translation illustrates that Java is a huge target space, a portion of which must be selected for the task in hand. The survey of six languages, two libraries and four Java extensions serves to put this point in context.

2 Approaches to translating to Java

Much praise has been heaped on Java from the start, and the language has not been found wanting. Specifically, older and specialised languages wish to ratchet up on Java to achieve portability and web-enablement, and hence a longer life for their existing programs. For example, a SmallTalk program that was previously used exclusively on one computer (for which the programmers had a compiler) could be recompiled through a SmallTalk to Java translator and sent around the world to any other computer. Similarly, libraries such as LAPACK which are written in Fortran, can, after automatic translation to Java, be incorporated into the whole new community of Java programs. In this instance, everyone wins: the LAPACK designers, because their product is more readily available, and the Java programmers, because they are getting tried and tested software to use immediately. It is also recognised that the Java community is now huge and growing, and that a language that wishes to make an impact can only improve its base by having a version which has Java as an intermediate language [Hardwick and Sipelstein 1996].

Nevertheless, there are various approaches to achieving Java-hood, as shown in Figure 1. Source to source translation (1a) involves producing valid Java source code, and many projects strive to make the code readable and as Java-like as possible. Java is not intended to be merely a machine-readable intermediate form. The reason is that taking advantage of the further power of the language, such as networking and graphics, will require that additional classes are written, in Java, to integrate with the translated source. Moreover, having a readable Java version of the program aids the clarity of the translation process.

The nature of the source-to-source translator can range from a completely new compiler to a minimal back-end that works from a syntax tree. This is the approach we took with Darwin. The advantage of working with a back-end is the speed with which the system can be developed, with measurements quotes in days, rather than months.

The second approach (1b) is an interesting one, and is taken by the NESL [Hardwick and Sipelstein 1996] and Eiffel [Potter 1997] projects. Both of these already produced interpretive intermediate codes (VCODE and Eiffel bytecode) and they chose to go up from this form to Java and then down again, even though it extended the compilation process. The reasons were that the NESL team did not want to adjust the front-end of the compiler which produced the VCODE; and the Bruce team felt that the structure of the Eiffel classes would be preserved if they worked this way.

The third approach (1c) is the simple one of taking a language down to bytecode, and is not discussed further in this paper.

The fourth approach (1d) is promoted by a mature project, PolyJ, which is now at Cornell, but took over work begun at MIT in 1997 [Myers *et al* 2000]. PolyJ programs can be translated directly to bytecode, or to Java source code, or as a combination of both. In other words, some classes can be in PolyJ and some in Java, providing considerable flexibility for developers.

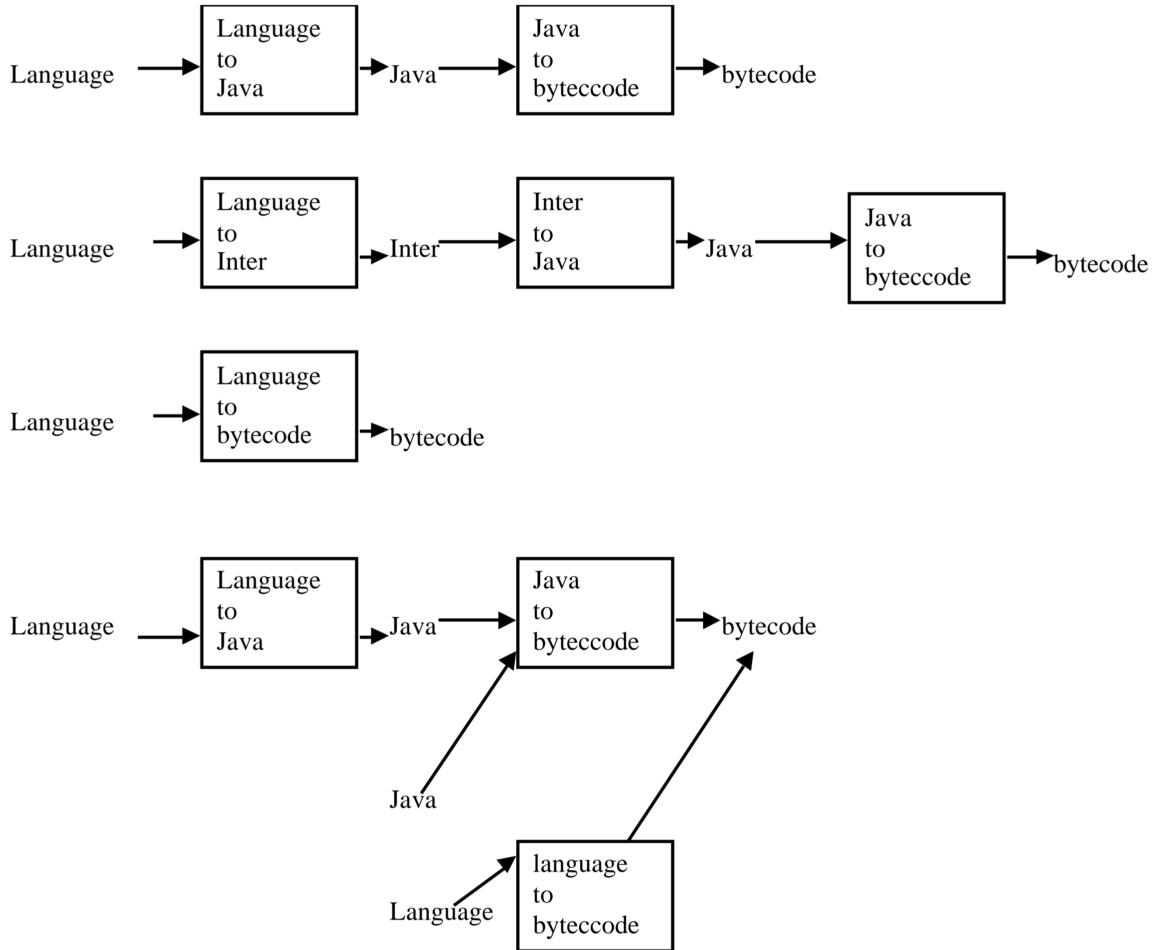


Fig. 1. Approaches to translating to Java bytecode (a) Source-to-source (b) Intermediate code-source (c) Source-to-bytecode (d) Hybrid

3. Issues in source-to-Java translation

Table I summaries twelve existing source-to-Java translators. There are, and have been, more, but these represent a representative set of those that are important and/or ongoing. The reader should note that many projects that mention translation to Java actually mean translation to bytecode, and these have been specifically excluded. The list includes six languages, two libraries, and four experimental systems aimed at adding extensions to Java.

We can now summarise the experiences of these projects and look at the issues they raise.

3.1 Helpful Java features

Across the board, the following were regarded as good points of Java that could be exploited in translation:

1. strong typing
2. isolated memory management and garbage collection
3. Java dynamic method dispatch
4. interfaces
5. portable GUI library (the awt)
6. network library
7. applet package
8. exceptions

Language	Project and Place	Language type	Translation type
LLP (Prolog superset)	Prolog Cafe at Nara NCT and Kobe U	Prolog with linear logic	source to source
Eiffel	Project Bruce at Microsoft and Macquarie, Australia	object-oriented language with generics and assertions	source to intermediate
Pascal, Oberon, Modula-2	Mill Hill and Canterbury Corporation	imperative languages in the Pascal family	source to source
NESL	Computer Science, Carnegie Mellon U.	high-level nested data-parallel language	source to intermediate
JLAPACK and BLAS	nba	numerical Fortran subroutines	source to source
PolyJ	PolyJ at Cornell U.	extended Java for polymorphism	hybrid
JASS	Theoretical Informatics Semantics Group, U. Oldenburg, Germany	design by contract in Java	source to source
SEAL	ASAP at U. Purdue, Geneva and Vienna	extended Java with tuples and closures[1]	source to source
Pizza	U. Karlsruhe and U. Glasgow	extended Java with parametric polymorphism, higher-order functions and algebraic data types	source to source
Darwin	Polelo at University of Pretoria[2]	ADL for configuring distributed components	source to source

Table I Summary of source to Java projects

[1] SEAL also had anonymous objects and iterators, which are now available in Java

[2] Based on a front-end from the Distributed Systems Group at Imperial College

It is actually up to the translator how plain or how fancy the resulting Java is going to be. With a long history of C target translations behind us, it is possible that features of a language will be implemented by simple method calls, without resorting to Java's rich inheritance, interface and abstract class structure. Some projects made a virtue of simplicity. For example, the PolyJ team was eventually able to implement their translation without using the Java reflection API or extending the JVM. This point is taken further in 3.3. The NESL team, which uses the intermediate-to-Java approach ended up losing semantic information for the translation input, and therefore were not able to take advantage of many of Java's features. The generated code consists mainly of calls to vector methods.

3.2 Deficiencies in Java

Three of the projects (PolyJ, Pizza and Eiffel) had to work hard to implement polymorphism or multiple inheritance in Java. The extensions proposed by PolyJ and Pizza essentially add a `where` clause to an interface definition thus [Myers *et al* 1997]:

```
interface SortedList [T]
  where T { boolean lessThan (T t); } ... { }
```

In PolyJ, `SortedList` would be instantiated for each of the classes for which objects are to be sorted and the signatures would be checked. The closest to this in Java is

```
interface Sortable {
  boolean lessThan  (Sortable b);
}
```

In Java, the interface name `Sortable` provides the link from the objects to a sort method. In other words, any class that declares that it implements `Sortable` will be accepted. Interfaces are not subject to the same stringent checking as would be required by polymorphic types, nor do they allow anything other than methods.

There are three approaches to implementing `where` clauses. One is to extend the JVM with opcodes to support the invocation of methods that correspond to where clauses (or signatures). The second produces a new class for each instantiated interface, but can quickly lead to code bloat. The third approach produces one class based on `Object`, and then inserts appropriate type casts wherever the class is used. If we want to run with a plain JVM, then the second is more efficient on time, the third on space.

Another problem that most translators address is multiple inheritance. Where a program does use multiple inheritance (as would be the case in Eiffel) then those parts can be flattened out [Potter *et al* 1997].

3.3 Accessing Java features

One problem is how to enable the programmer to take advantage of features in Java for which there is no parallel in the existing language. Two approaches are evident. The first is that taken by PolyJ which enables mixed Java and PolyJ programs. The difficulty here is in the name space. The translated PolyJ must create names that do not conflict with names chosen later in the Java classes.

The second approach is to provide stub classes in the language for those Java classes that the programmer should know about. For example, Eiffel has stubs for the applet class. The Eiffel programmer can redefine its methods in the normal way.

4. Overview of Darwin and Regis

Darwin is an architectural description language which enables components written in any language to be connected together in a configuration and distributed over different machines [Magee *et al* 1995]. Darwin is declarative in nature and includes facilities for parameterised components and interfaces, generic types, dynamic creation of components and limited control structures (for, if). An example of a Darwin program is:

```
component Sorter(int inputCount) {
             @ processor("137","215","18","41") {
    require portal p: <T1>;
    provide q: fileIO(3);
    inst X : sorter(13, <T1>);
    inst y [13] : sorter @ processor(10);
    when fullInput == true {
         bind X.output1 -- y[13].input1;
    }
```

Here the `Sorter` component instantiates 14 `sorter` components (not shown) – one called `x` and an array of 13 y's.

Regis is the runtime environment on which the resulting configured system runs. A program running on Regis consists of a tree of components of which the root and all non-leaf nodes are composite components, written in Darwin. Leaves are primitive, algorithmic components written in any language, such as Java or C++ or Fortran. While such programs are closed systems, there is support for inter-application communications.

Regis provides several services to components. They include:

1. A set of communication objects, facilitating message passing between components;
2. A standard library of pre-programmed components
3. Remote procedure call functions
4. A naming service
5. A remote execution daemon

Figure 2 illustrates the relationship between Darwin and Regis. Our intention was to replace the backend of the compiler so that it produced Java, and to rewrite Regis as Jade, in Java, and using Java's network support facilities.

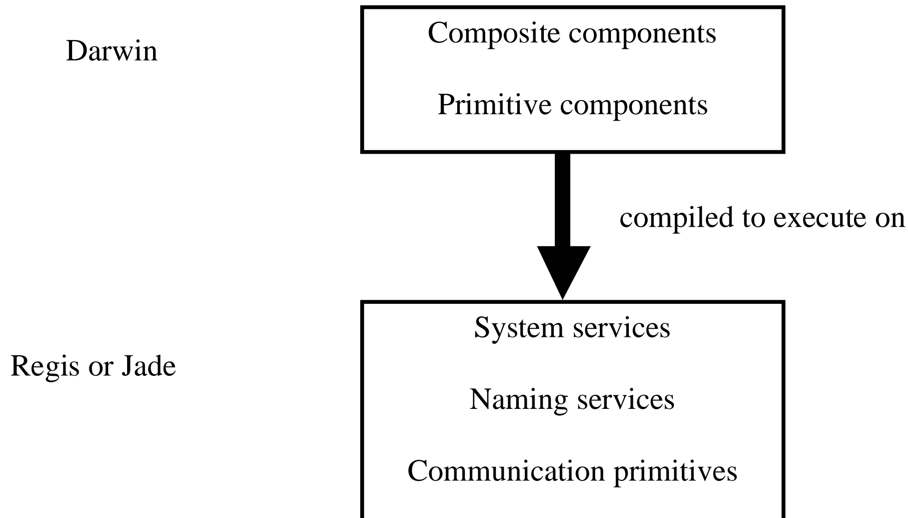


Fig. 2. Architectural view of Darwin and Regis/Jade

A component in Regis is seen as a single thread in a Unix process. Components in Regis are derived from the `Process` base class which controls the instantiation and distribution of distributed components. Components may be distributed arbitrarily on any of the participating machines. The Regis runtime environment provides a remote execution daemon to achieve location transparency to the components.

The Regis Execution Daemon (RED) provides services for the remote execution of objects, creating a large virtual parallel machine. Each virtual host has a unique identification number. RED maps these numbers to a set of real Unix machines using an internal algorithm or by using user preferences. Therefore a physical host may have several virtual hosts on it at any time. Each physical machine taking part in the Regis virtual machine needs to have a RED running.

Components send and receive messages on port objects. In Regis, ports use the C++ template facility: before a port can be used, it must be parameterised with the type of data that the port can receive. Messages are sent from `Portref` objects to `Port` objects. Before a component can use a `Portref` to send messages to another component, the sender's `Portref` needs to be bound to the receiver's `Port` object. All operations on the sender's `Portref` will block if is not bound to a port.

The next section describes the work that has been done in creating a Java-based runtime system with functionality similar to Regis (Jade, or JAva Distributed Environment) and the Darwin backend that was created to produce classes that will run on Jade.

5 The Darwin to Java translator

5.1 Design philosophy of Jade

The main idea in designing the distributed environment is to provide the same functions that Regis currently provides, but in the Java programming environment rather than the Unix/C++ environment on which Regis

is based. Even though Java is a general-purpose programming language, the style of programs written in Java is radically different from other languages. Because Java is interpreted and tightly integrates the network into the programs, the overall design of programs is often quite different from their C++ counterparts. A straightforward porting of Regis from a C++ and Unix-based platform to a Java-based platform which does not rely on Unix system services is not a feasible option. The approach followed in this project was to take the functionality of Regis and implement that in Java without much regard for the internal structure of Regis. Jade's implementation does, however, make full use of the features mentioned in the previous section, to enable us to create a simple structure while at the same time obtain as good performance as possible.

Figure 3 is a class diagram of Jade in UML notation. There are two main groupings of classes. The first grouping concerns processes. Each process that runs in Jade is a subclass of `RemProcess`. The Remote Execution Daemon (RED) is also a subclass from the `RemProcess` class, so in effect RED is just another user-level program.

The second grouping concerns the mechanisms for message passing between processes. The `Port` and `PortRef` classes have the same meaning as they have in Regis. There are some support classes shown, such as the `REDMessage` group of classes. These classes and their use will be mentioned when they are used by the other classes.

5.2 Differences between Jade and Regis

All communication in Jade is done via RMI. This changes the system in important ways:

- **Name service** Regis had to rely on its own primitive name server, `RND`. Jade uses the RMI naming service instead. The result is that there is no Regis Name Daemon and no `nameserver` class to act as a wrapper. All naming lookups are done by static methods in the various classes.

- **Message passing** Regis used its own communications mechanism based on UDP packets. Jade uses RMI objects and therefore it uses the RMI message passing system. The big advantage is that one can transfer data structures and objects between processes, making complex object interaction possible.

- **Generics** Java does not have support for generics. Regis makes extensive use of templates in order to provide typed ports and messages. It was therefore necessary to slightly modify how ports and messages are created and used.

- **Processes** Processes in Jade are single threads of executing objects. All processes inherit from `RemProcess` which corresponds to class `process` in Regis.

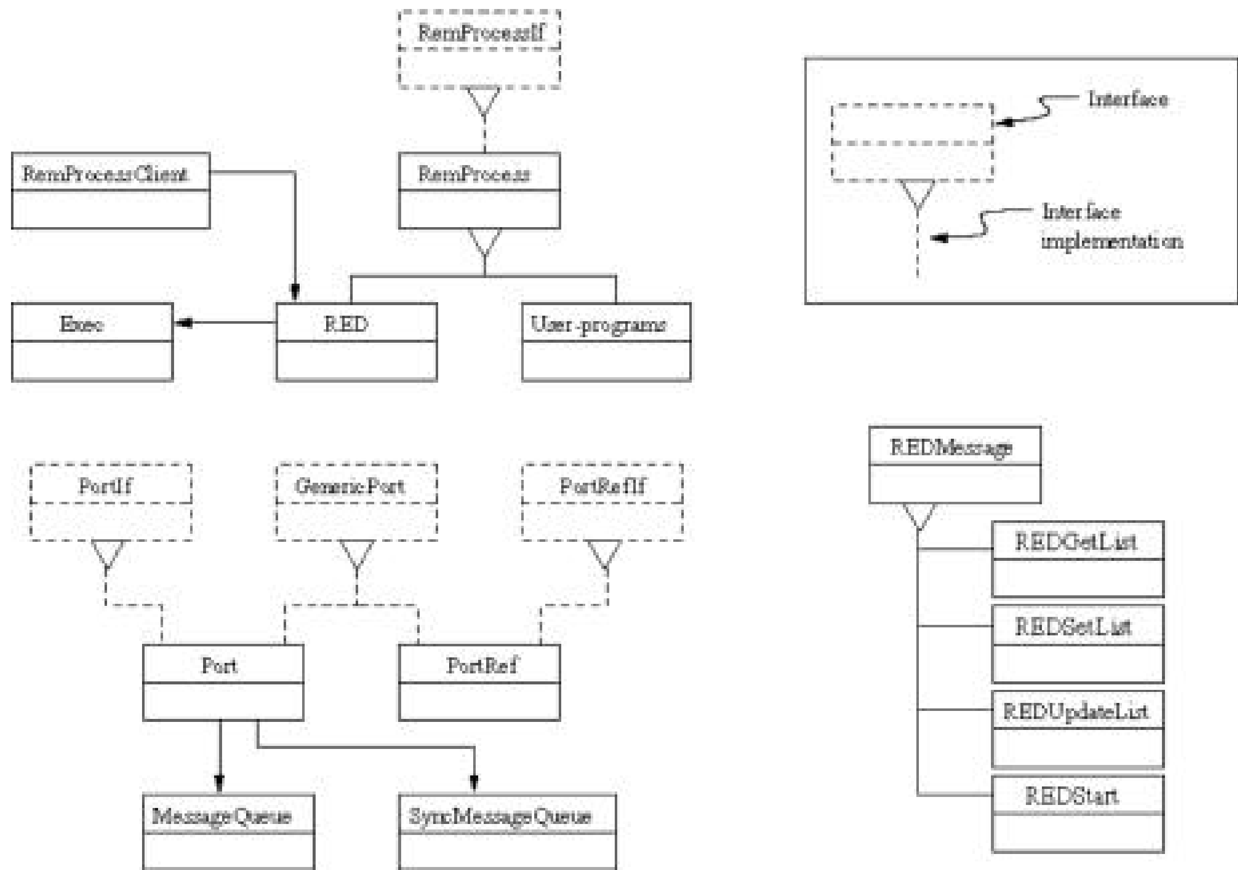


Fig. 3. Architectural view of the Java Distributed Environment

```
public class RemProcess extends UnicastRemoteObject
       implements RemProcessIf, Runnable {
   public RemProcess(RED red, String name, Hashtable args);
     // Registers the process as service 'name'.  args contains
     // the optional arguments of the component.

     public void go();
     // Starts the process running in the background. Calls run().

     public void run();
     // All processing is defined in run()

     public static RemProcessIf resolve
               (String host,String component);
     // Does a name lookup for a specific service

}
```

As alluded to earlier, the structure of a Jade process is rather different to its Regis counterpart. In Regis, all computation is put in the constructor of the `process` class, so that the Regis scheduler could handle scheduling properly. Because all Jade processes are required to implement the `Runnable` interface, it follows that all processing is done in a `run()` method.

The constructor of class `RemProcess` performs the action of registering the process with the RMI name server. From that point onwards, any other process can get a remote reference to it and use it for its own purposes. Unlike Regis processes, all Jade processes are instantiated using the RED server. To that end, there is a utility program called REDLaunch, which allows one to start a remote process. Here is a HelloWorld program in Jade. This would be the output from a Darwin-to-Java compiler.

```
class hello extends RemProcess {
     public hello(RED red, String name, Object[] args) {
          super(red, compname, args);
     }
     public void run() {
          System.out.println("Hello World");
     }
}
```

5.3 Communication between components

`Ports` and `PortRefs` have the same meaning as in Regis. They perform the same functions and provide the same services. Both the `Port` and `PortRef` classes are full RMI services. They are registered in the naming service and can be accessed by anyone through the `resolve()` methods in the relevant classes.

Ports may be typed or untyped. They are by default untyped, which means that any can be transmitted between them. It is also possible, by specifying a class type in the constructor, to restrict ports to receive messages of only one type. All messages must be objects; this means that all primitive types (`char`, `int`, `boolean` etc) have to be wrapped in their Java object wrappers (`Character`, `Integer`, `Boolean`, etc) before being sent between components.

Port references are represented by the `PortRef` class. They are initially unbound and untyped. When a port reference is bound to a port with the `PortRef.bind()` method, it takes on the type of the `Port` object to which it binds.

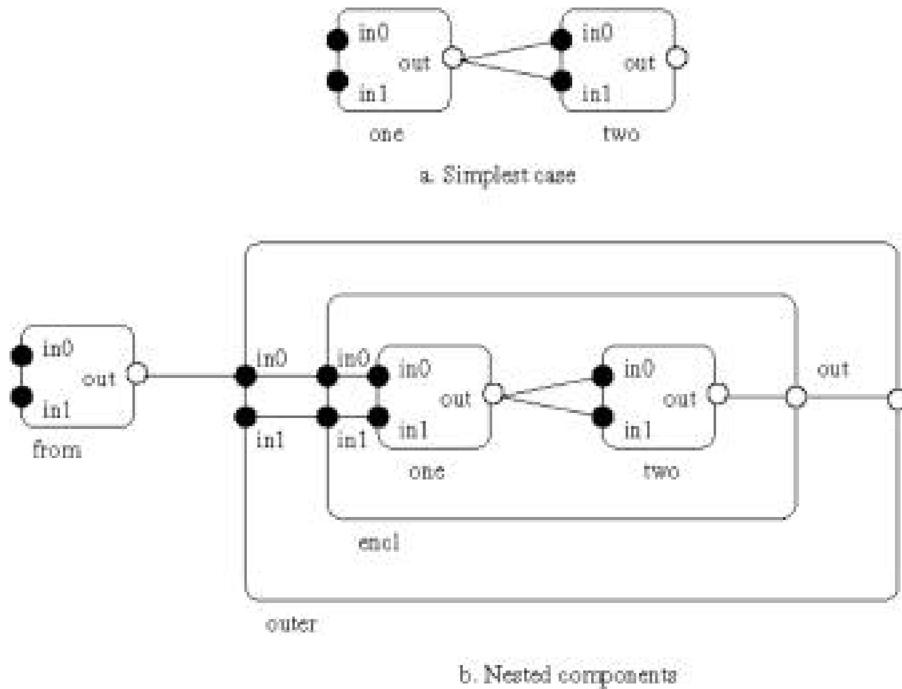
5.4 Binding of ports

The process of binding together of ports is rather complex and therefore requires some explanation. The simplest case of the binding is when a `PortRef` object is bound to a port object and both `Port` and `PortRef` are ports of two primitive components that are not part of any composite component. This situation is shown in part (a) in figure 4. In this case a call to `one.out.bind("two","in0");` will bind the `PortRef` called `one.out` to the port called `two.in0` without any trouble.

The situation gets more complex as soon as a component is encapsulated by another one, as in part b in figure 4. The problem is that nested components form a sequence of port objects all bound to each other, from the outermost component down to the innermost one. Similarly, a sequence of `PortRef` objects

appears, running from the innermost component to the outermost one. In our example, should a message arrive at `outer.encl.in0`, the message must pass through two intermediate ports before it reaches `encl.one.in0` where it is finally used. In even moderately complex systems, the number of intermediate `Ports` and `PortRef` quickly becomes unwieldy.

A solution to this problem is to flatten the component structure in order to remove all intermediate (extraneous) interfaces. Figure 5 shows how the structure of our example can be flattened. From the diagram it is clear that a portref can then communicate with a port across many levels of nested components without the intermediate components' ports generating unnecessary communications overhead. The flattening of the component structure is automatic and completely transparent to the user and does not affect the way programs are written in any way.



6 Binding of PortRefs to Ports

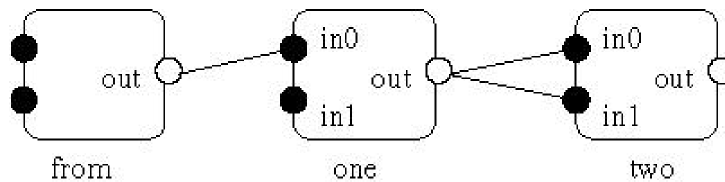


Fig. 4. Flattening the components

6 The Darwin to Java compiler

The Darwin compiler has three phases, as in any traditional compiler. The syntactic and semantic analysis phases and the code generation phase are well separated as shown in the calls to the various phases of the compiler.

```
1   JavaBackEnd jbe = new JavaBackEnd();
2   DarwinCompiler compiler = new DarwinCompiler(this, jbe);
3   Vector components = null;
4   try {
5       components = compiler.parseComponents();
6       compiler.checkSemantics(components);
7       compiler.generate(components, this);
8       outerrln("Success");      // Show success
9       this.close();             // Close the output file
10  }
11  catch (DarwinException e) {
12      int i = ((Integer) (e.marker)).intValue();
13      outerrln("ERROR - " + e.getMessage());
14  }
```

In the syntactical analysis phase (line 5), the source code is parsed and a source .syntax tree is constructed from the input. The second phase, the syntax tree is verified to conform to the Darwin language specification (line 6). Once the syntax tree is verified for semantic correctness, it can be converted to Java source code. That happens in the code generation phase (line 7). The `JavaBackend` class performs the code generation phase and contains all the compiler-related code that was written during this project.

6.1 The source syntax tree

During the parsing phase, a syntax tree is created. This syntax tree is a representation of the contents of the input source file, which can be manipulated by the compiler and the backend to generate the target code. Each type of declaration in the Darwin language has a class representing it. Thus we have `ComponentDeclaration`, `InterfaceDeclaration`, `PortalDeclaration`, `ForAllDeclaration`, `WhenDeclaration`, etc. All these classes inherit from the `Declaration` base class. The syntax tree is a tree consisting of these `Declaration` classes. This tree is given as input into the Java backend. The code generator takes the tree and traverses it, producing code as it goes along. Each `Declaration` subclass contains all information that is relevant to that declaration, including all the necessary state that is required to implement that declaration.

This paper can only give an overview of the algorithms involved and the structure of the code generator. The full code for the code generator (class `JavaBackend`) is available on the web at www.cs.up.ac.za/~jbishop/Jade and is very well documented.

6.2 Mapping Darwin onto Java

This section looks at the main lexical features of Darwin and how they are supported in the Java backend. Here we shows a Darwin input file and one of the Java source files that was generated by the compiler. The full example is in Botha [1998] and on the website.

```
interface int4
{int0; int1; int2; int3;}

interface int2(int name, boolean boolvalue)
{int0;int1;}

component Switch(int AA) {
     require in: int2(AA, false);
     provide out: int2(AA, true);

     when AA == 1 {
          provide backdoor;
     }
}

component Sort2(string A, int B, boolean C) {
     require in: int4;
     provide out: int4;

     boolean istrue = true;

     inst
          Switch1: Switch(2);
          Switch2: Switch(B);

     when A == "exp" {
          inst Switch3: Switch(1);
     }

     bind
          in.int0 -- Switch1.in.int0;
          in.int1 -- Switch1.in.int1;
          in.int2 -- Switch2.in.int0;
          in.int3 -- Switch2.in.int1;
          out.int0 -- Switch1.out.int0;
          out.int1 -- Switch1.out.int1;
          out.int2 -- Switch2.out.int0;
          bind out.int3 -- Switch2.out.int1;

     when A == "exp" {
          bind in.int0 -- Switch3.in.int0;
     }
}
```

Predefined types All the predefined types are supported. The `string` type is translated to its Java type, `String`. When parameters must be passed during component instantiation or interface instantiation, all types are cast to their object wrappers. This is necessary because all parameters are objects, so all primitive

types must be formatted to fit in with this scheme. `int` becomes `Integer`, `boolean` becomes `Boolean`, `string` becomes `String` and `double` becomes `Double`.

```
import java.util.*;
import darwin.*;
import java.rmi.*;
import java.io.*;
// This code was generated by the D2J compiler

class int2
{
      GenericPort int0;
      GenericPort int1;

      public  int2(String  parent,  String  type,  Integer  name,  Boolean
boolvalue)
      throws RemoteException
      {
            if(type.equals("Port")) {
                  int0 = new Port(parent,"int0",null);
                  int1 = new Port(parent,"int1",null);
            } else {
                  int0 = new PortRef(parent,"int0");
                  int1 = new PortRef(parent,"int1");
            }
      }
}
```

Component declarations The generation of components is done in the `JavaBackEnd.genComponent()` method. At the start of each component, it opens a new file for the component and adds a set of import statements. The variable list follows. This includes declaring all the ports, portrefs and interfaces used in the component. It takes into account the ports that are conditionally created.

The constructor is then created. After some initial standard calls to the constructor of the class' superclass, it prints out a list of the types and the names of all the formal parameters that have been passed to the component in a hashtable. This is for reference for the programmer, should he want to use them.

Next comes the instantiation of all the sub-components of that component. Actual parameters are sent to these sub-components, by creating a hashtable for each sub-component and adding the parameters to it. Sub-components that are conditionally instantiated (with a `When` statement) are also constructed here, including the condition that controls the instantiation.

All the ports and portrefs are instantiated next. Interfaces are also instantiated here. Once all the interfaces, ports and portrefs are instantiated, they can be bound to the various sub-components. Bindings for all the ports are created at this point.

Portal declarations The `genPort` and `genPortRef` methods take care of all portal declarations. This includes interface declarations. Typed portals are supported, but by default portals are untyped. Darwin uses generic types to provide types to portals. As stated earlier, generics are not supported. Even though the `Port` and `PortRef` classes are typed and can be parameterised with a message type, the current compiler

produces only typeless code. Portal arrays are not supported, for the same reason that instance arrays are not done.

Interface declarations Parameterised interface types are supported and are generated in the `genInterface` method. As explained in the previous part, `Ports` and `PortRef` are two different classes. This causes a bit of a problem when creating a class definition for an interface, because at the interface declaration time, it is not known whether the interface contains `ports` or `portrefs`. The solution to the problem is this: both the `Port` and `PortRef` classes implement the `GenericPort` interface. When the interface is declared, all its member portals are declared as of type `GenericPort`. The constructor has a parameter that specifies whether the interface is being used as a set of `Ports` or a set of `PortRefs`. Therefore, when the interface is instantiated, its portals are cast into either `Ports` or `PortRefs`.

Other lexical features Constant declarations, external declarations and tag declarations have not been implemented. All of these pose significant problems and would require some serious attention before they can be implemented.

Constant declarations are relatively straightforward to implement. In C++, it is easy to do, since they can be implemented as global variables, which are visible to everyone automatically. Java, on the other hand, has no support for global variables and as such a workaround must be found. One way is to create a special class that contains all the constants. Some problems with this approach are the naming of such a class (what if we have more than one Darwin program that runs simultaneously? The naming service will get confused if we give the constants class a static name, but if we don't give it a static name, how will the other classes know what the name of the constants class is?) and how to control the scope of the constants (related to the first problem).

6.3 Limitations

The features that have not been implemented are partial component declarations, generic types (templates), assertion declarations and dynamic instances. These may be added at a later stage. However, more time is required than what was available for this project to fully handle those topics. The compiler's instance array handling is not finished. Therefore instance arrays have not been implemented, and thus `Forall` declarations are not implemented either, since the two features are tightly linked.

The implementation of external declarations will require more thought. External declarations allow the creation of declarations that entities outside Darwin define. The external declaration model for a Java-based program will have to be thought out carefully. Because the lexical structures of Java are so different from C++ (the absence of generics, etc), it will necessitate a radically different approach to external declarations.

Tag declarations also pose a rather tricky problem. Tags can be associated with almost any other declaration in Darwin. Their parameters are not fixed and are programmer-defined. There is no formal structure to them. Such a feature is rather easy to implement with C or C++, because of the language feature of C++ that allows variable length parameter lists to a method. Java does not provide such functionality, and as such the structure of tags will have to be thought through carefully to determine how the same functionality may be provided in maybe a slightly different format.

7 Conclusions

The Darwin to Java translator was not hard to accomplish and performs without errors. The runtime environment on which it operates uses RMI and the performance has been shown to be the same, if not better, than that of the C++ Regis.

Further work is now proceeding on completing the compiler and on investigating other forms of communication for Regis. The newer APIs for serialisation and for CORBA can now be incorporated.

And finally, the most cited paper in this area [Odersky and Wadler 1997] begins with the quote:

There is never anything new under the sun. *Ecclesiastes 1:8*

So I would like to end with:

Ex Africa semper aliquid novi Pliny, *Historia Naturalis*

Acknowledgments

This work was support by the National Research Foundation. I would like to acknowledge the work done by Louis Botha on the compiler, and the assistance of Jeff Magee, Jeff Kramer and those involved in the early port of the Darwin compiler at Imperial College.

References

Banbara M and Tamura N, Translating a linear logic programming language into Java, Proc. ICLP'99 Workshop on Parallelism and implementation technology for constraint logic programming languages 19-39, Dec 1999 and http://c32.scitec.kobe-u.ac.jp/~banbara/PrologCafe.html

Bartetzko D, The Jass page, http://semantik.Informatik.Uni-Oldenburg.DE/~jass, 2000

Botha L J, Java in configurable distributed systems, Honours project, University of Pretoria, 1998, also on http://www.cs.up.ac.za/~jbishop/Pubs/Botha.ps

Bothner P, Kawa: Compiling Scheme to Java, www.mit.edu/afs/sipb/project/kawa/doc/kawa-tour.html 1997

Boyd N, Bistro = Smalltalk over Java, http://www.jps.net/nikboyd/papers/bistro/bistro.htm 1999

Doolin DM, Dongarra J, Seymour K, JLAPACK- compiling LAPACK Fortran to Java, http://www.cs.utk.edu/f2j/f2jreport/f2jreport.html 1998

Hardwick J C and Sipelstein J, Java as an intermediate language, CMU-CS-96-161, 1996

Krall A and Vitek J, On extending Java, Proc of the Joint Modular Languages Conference, Linz Austria, March 1997

Magee J, Dulay N, Eisenbach S and Kramer J, Specifying distributed software architectures, Proc 5th European Software Engineering Conference, , Barcelona Spain, Sep 1995

Mill Hill & Canterbury Co, Java programming, wysiwyg://125/http://www.webcom.com/mhc/java.html 2000

Myers A C, Bank J A and Liskov B, Parameterised types for Java, Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997 and http://www.pmg.lcs.mit.edu/polyj.html 2000

Odersky M and Wadler P, Pizza into Java: translating theory into practice, Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997

Potter J, Noble J, Shelswell R, Project Bruce: translating from Eiffel to Java, Eiffel Liberty site, http://www.progsoc.uts.edu.au/~geldridg/eiffel/liberty/ 1997

Taft, T, Programming the internet in Ada 95, Proc Ada Europe, 1996

Tolksdorf, R, Programming languages for the Java Virtual Machine, http://grunge.cs.tu-berlin.de/~tolk/index.html 2000