Integration of BETA with Eclipse – an exercise in language interoperability

Ole Lehrmann Madsen, Peter Andersen, Mads Brøgger Enevoldsen [1,2]

Department of Computer Science
University of Aarhus
Åbogade 34, DK-8200 Århus N, Denmark

Abstract

This paper presents language interoperability issues appearing in order to implement support for the BETA language in the Java-based Eclipse integrated development environment. One of the challenges is to implement plug-ins in BETA and be able to load them in Eclipse. In order to do this, some form of language interoperability between Java and BETA is required. The first approach is to use the Java Native Interface and use C to bridge between Java and BETA. This results in a workable, but complicated solution. The second approach is to let the BETA compiler generate Java class files. With this approach it is possible to implement plug-ins in BETA and even inherit from Java classes. In the paper the two approaches are described together with part of the mapping from BETA to Java class files.

Key words: Language interoperability, BETA, Java, JVM, Eclipse, debugger, .NET, CLR.

1 Introduction

The goal of this paper is to present experiences from a project on implementing support for the BETA [11] language in Eclipse [2]. Eclipse is a general

[1] Email:{olm,datpete,brogger}@daimi.au.dk

[2] The integration of BETA with Eclipse has been sponsored by Eclipse.org[2] and the actual work has been carried out as part of the Master Thesis project by Mads Brøgger Enevoldsen [3], co-supervised by Peter Andersen and Ole Lehrmann Madsen. The porting of BETA to CLR and integration with Visual Studio has been sponsored by Microsoft Denmark and Microsoft Cambridge. The porting of BETA to JVM has been sponsored by Sun Microsystems. The porting of BETA to JVM and CLR has been carried out by Peter Andersen and Ole Lehrmann Madsen. We gratefully acknowledge the support, inspiration and help from Jakob Roland Andersen, Lars Bak, Brian Berry, Gilad Bracha, Susanne Brøndberg, Neil Gafter, Kim Falk Jørgensen, Henry Michael Lassen, Kasper Verdich Lund, Henrik Lykke Nielsen, Dave Thomas, and Jørgen Thyme.

Integrated Development Environment (IDE) with a plug-in architecture that facilitates addition of new tools. Eclipse is implemented in Java[4] and the architecture is well prepared for writing plug-ins in Java. It is possible to write plug-ins in other languages than Java, but this requires considerably more work. The main subject of the project has thus been to investigate language interoperability issues appearing in order to implement support for BETA in the Java-based Eclipse IDE.

BETA is an object-oriented programming language implemented on a number of platforms with native compilers. For program development, a powerful IDE – the MjolnerTool – is available as part of the Mjølner System[9]. The MjolnerTool is well suited for program development in BETA. There are, however, a number of reasons for investigating the suitability of using a general IDE such as Eclipse:

- Most software developers use several programming languages depending on the kind of project they are working on. Using the same IDE for all the languages will make it more efficient to alternate between languages.
- Libraries and frameworks may often be used from another language than they are implemented in. So far language interoperability between object-oriented (OO) languages has been limited. In the simplest form, procedure libraries in e.g. C may be used from an OO language. Using Microsoft COM[5], it has been relatively easy to use components (and objects) across languages. With Microsoft's .NET[6]/CLR[7] [3] it has been demonstrated that it is possible to reuse classes between languages. If language interoperability increases, it is of course an advantage to be able to support several languages from the same IDE.
- A general purpose IDE should be able to provide more tools than a special purpose IDE.
- It requires considerable resources to support an IDE like the MjolnerTool. Using a general IDE, the maintenance burden will hopefully be reduced to the specific tools being offered by e.g. BETA.

There are a number of requirements that need to be fulfilled in order to replace the MjolnerTool with Eclipse:

(i) All (or at least most of) the tools available in MjolnerTool should be available in Eclipse. These include code browsing, text-editing with indentation, syntax-directed editing, semantic browsing, compilation, debugging, a UML-like CASE tool and an interface builder. Furthermore, other features of modern IDEs such as syntax colouring and semantic tool-tips should be available.
   (a) Code browsing, text editing, syntax colouring, and compiler invocation were expected to be straightforward to support in Eclipse.
   (b) Indentation, syntax-directed editing, semantic browsing, CASE tool

[3] CLR is short for *Common Language Runtime*.

and interface builder are all language specific and it was expected that these tools would have to be provided as new plug-ins.

(c) For debugging it was not clear from the beginning whether or not it would be possible to reuse some of the Eclipse functionality or a specific BETA debugger would have to be supplied.

(ii) Plug-ins for BETA should be implemented in BETA. The reason for this is that it will allow for reuse of existing BETA code from the MjolnerTool. Furthermore, if BETA developers and users would have to switch to Java in order to implement BETA tools, the motivation for using Eclipse will be smaller.

(iii) BETA libraries and frameworks should in general be usable together with Eclipse. For the non-GUI frameworks no problems were expected, but for the GUI libraries it is not at all clear to what extent this can be done when e.g. implementing Eclipse plug-ins.

Below two approaches for integrating BETA with Eclipse are described. The initial approach is to use JNI to bridge Java and BETA via C. In this way, it is possible, in principle, to write parts of the plug-ins in BETA. As said below, this turns out to be tiresome and very complicated to debug.

The second alternative is to generate Java class files from BETA and launch these class files in Eclipse. Given this, it should be straightforward to write plug-ins supporting most of the above requirements.

It is, however, a major task to generate class files from BETA. class files are designed to support Java and in many ways BETA is more general than Java. The reason why this was considered a realistic alternative was that the BETA.Eclipse project was carried out at the same time as a project for implementing BETA on JVM[4] and Microsofts CLR/.NET platform.

The goals of that project are to investigate the suitability of modern (typed) virtual machines for supporting a language like BETA, to evaluate JVM and CLR as virtual machine platforms, and to investigate whether language interoperability as claimed for CLR/.NET can work with BETA and also to what extent language interoperability is possible on JVM.

The status of the two projects are as follows:

(i) BETA has been ported to JVM and CLR.

(ii) Language interoperability for BETA on .NET seems to work as promised – the surprising thing (at least to the authors) is that it works just as well on JVM.

(iii) A simple form of integration of BETA with Eclipse has been obtained using JNI. This includes browsing, text editing, indentation, and launching of the compiler. The indenter module is written in BETA.

(iv) A much more tight integration has been obtained by using the BETA-

[4] JVM is short for *Java Virtual Machine*.

to-JVM compiler.[5] Plug-ins can be written in BETA, compiled to class files and launched in Eclipse.

(v) Using JBeta together with Eclipse it is possible do high-level debugging of BETA programs by means of the Java debugger available in Eclipse.

2 Initial Eclipse Integration

2.1 The Plug-in

Code-browsing and raw text editing of BETA source files is possible in Eclipse "out of the box". However, as Eclipse does not know about BETA source files it will be inconvenient to use. As an example, Eclipse will complain about not knowing the language for the file just opened and suggest to edit the file using a standard text editor.

By making a small Eclipse *plug-in* consisting of just one XML file, it is easy to specify that files with the `.bet` extension should be presented with a special icon and use the standard text editor without asking.

More functionality can then be added to the plug-in, by supplying specialized classes for e.g., text editing. Syntax colouring can be added with a few classes, whereas compiler activation requires implementing a larger number of classes since this typically involves extending the Eclipse tool-bar.

2.2 Adding Indent Strategy using JNI

For Eclipse text editing, a so-called *indent strategy* can be associated. This specifies a default behaviour for indentation of code. A default indent strategy called `DefaultAutoIndentStrategy` exists in the Eclipse class library. Implementing a specific language indent strategy is obtained by overriding methods in this class.

In the Mjølner System library there is an implementation of an indent strategy for BETA code. The goal is to use this existing BETA library to implement the indent strategy in the editor plug-in.

The JDK[6] library provides an interface to native methods called JNI[7]. This allows code running in the Java Virtual Machine to access library methods written in C. BETA also has a C interface, which combined with JNI makes interoperability between Java and BETA possible through C.

The implementation – illustrated in figure 1 – consists of three layers, deployed into a Java part and two external libraries. The Java part implements the Eclipse extension point. The library entitled `JniIndentWrapper` contains all the "plumbing" needed to call C from Java, whereas the library entitled

[5] From now on called JBeta
[6] Short for *Java Development Kit*
[7] Short for *Java Native Interface*

`CwrappedBETALibrary` contains the BETA code to be called by Java as well as a set of C functions exposing the BETA library to JNI.

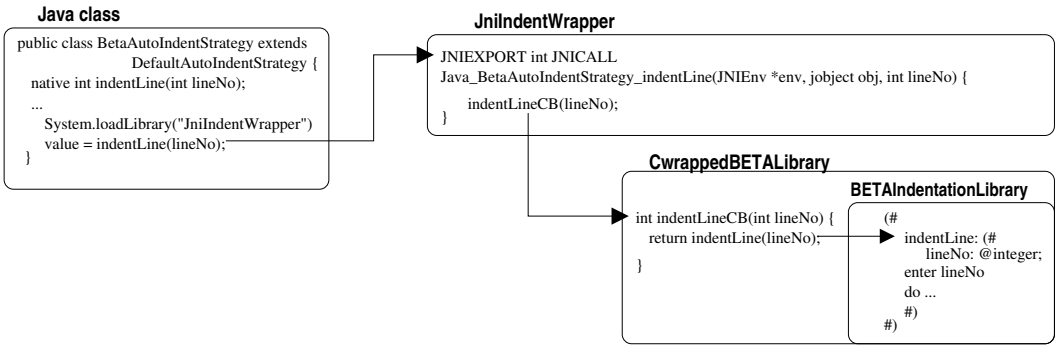

Fig. 1. Java BETA interoperability using JNI

This form of interoperability requires the BETA plug-in to be written in Java, but allows some part of the functionality to be implemented in BETA. This facilitates reuse of existing BETA code. The code for the Java class calling the BETA libraries through JNI is sketched below:

```
class BetaAutoIndentStrategy extends DefaultAutoIndentStrategy {
  native int indentLine(int lineNo); // External method
  public BetaAutoIndentStrategy() {
    System.loadLibrary("JniIndentWrapper"); // Load external library
    ...
  }
  public void customizeDocumentCommand(IDocument d,
                                       DocumentCommand c) {
     ...
     value = indentLine(lineNo); // Call external method
     ...
  }
}
```

2.3 Evaluation

Code browsing and raw editing are indeed available "for free" in Eclipse per se. And with a small plug-in specification it is possible to make Eclipse know about source files with a given extension. However, to obtain basic functionality such as syntax colouring and compiler invocation, quite some coding is needed. Perhaps more than was expected.

The implementation of the BETA indent strategy by reusing BETA libraries through JNI consists of more than 1000 lines of BETA code in several source files. This shows that this technique is possible to use for a non-trivial example. However, it has several disadvantages: One problem is that three different languages are used. This makes the integration tedious and complicated. Not only is it complicated to write the code, but more importantly

once written and compiled it is far from certain that it will work. Loading of the libraries happens at run-time and this late binding of the code makes it impossible for the compiler to make even simple syntactic and semantic checks on e.g. the various method invocations. When a run-time error occurs in external code there is little information from the JVM. Often one has to suffice with the error message `Crashed somewhere outside the VM`. Debugging such a bug is done by carefully looking at the source code and using print statements everywhere!

Overall, this solution is far from good. It solves the problem but for the implementor, in a very inconvenient way. The next sections describe a different approach that solves the problem in a native way – namely by compiling the BETA source *directly* to JVM bytecode, thus eliminating the need for the JNI layer.

3 Implementation of BETA on Java Virtual machines

Before presenting the JBeta solution for Eclipse integration, parts of the mapping of BETA to JVM are presented – see [1] for a complete description. There have been two overall issues regarding the mapping of BETA to JVM:

(i) JVM is a typed virtual machine designed to support Java and it was in no way obvious that BETA could be mapped to JVM. One overall issue has thus been whether or not it was possible to find a mapping to JVM.

(ii) The other issue is language interoperabiliby as seen from the programmer's point of view. To obtain true language interoperability, it should be possible to use classes written in Java from BETA and vice versa. Given an implementation of BETA for JVM this should be possible. The main issue will be readability/understandability of doing this. Using Java from BETA will probably not pose problems, since a BETA programmer just has to understand the Java classes being used.

The other way around may be more problematic, since BETA is more general than Java. Java programmers that will use BETA should be presented with a view of BETA as Java classes reflecting the mapping of BETA to JVM. This means that a *readable* mapping must be defined and this is of course a stronger requirement than just finding a working mapping.

Note that this paper is not an attempt to present the rationale behind the generality of BETA. The purpose is to present some of the issues in mapping BETA to JVM. This will be illustrated by more or less useful examples. The rationale behind BETA has been presented in a number of other places such as [10,11]. A short summary of BETA is given in the Appendix.

3.1 Example of a BETA program mapped into Java

BETA has one major abstraction mechanism called *pattern*. A pattern unifies, e.g., the concepts of *class* and *method* known from other languages like Java and C#[8].

Consider the following example of a BETA pattern describing a calculator that may add a sequence of values:

```
Calculator:
  (# R: @integer;
     set: (# V: @integer enter V do V -> R #);
     add: (# V: @integer enter V do R + V -> R exit R #)
  #);
```

The `Calculator` pattern has three attributes `R`, `set` and `add`. `R` is an instance variable holding the current value of the `Calculator`. The attributes `set` and `add` describe operations on the calculator. The operation `set` is used to set the value of `R` to the value of the enter-parameter `V`. The operation `add` adds the value of the enter-parameter `V` to `R` and returns (exit) the value of `R` to the caller.

The following example shows an instance `C` of `Calculator`, an instance `X` of `integer`; a call of `C.set` with enter arguments `12`; and a call of `C.add` with enter argument `5` and the resulting exit-value being assigned to `X`.

```
C: @Calculator; X: @integer;
12 -> C.set;
5 -> C.add -> X;
```

Note that `Calculator`, `set` and `add` are all examples of patterns.

3.2 A simple but incomplete mapping

In the example in the previous section, `Calculator` is used as a class and `set` and `add` as methods. This use of the `Calculator` pattern is illustrated by the following simple mapping to a Java class:

```
class Calculator extends Object {
   int R;
   void set(int V) { R = V; }
   int add(int V) { R = R + V; return R; }
}
```

The BETA declarations and invocations shown above then map to the following declarations and invocations in Java:

```
Calculator C = new Calculator(); int X;
C.set(12);
X = C.add(5);
```

3.3 A complete mapping

The above mapping shows a simple semantics of the BETA `Calculator` pattern. To capture the full semantics of the `Calculator` pattern a more complex mapping is needed. The fact that e.g. `add` is a pattern means that it is possible to use `add` as a class and create instances of `add` as in the following example:

```
C: @Calculator; X: @integer;
A: ^C.add;
&C.add[] -> A[];
```

The variable `A` may refer to instances of `C.add`. The statement `&C.add[] -> A[]` creates an instance of the pattern `C.add` and assigns its reference to `A`. It is now possible to use `A` to assign a value to the instance variable `V`:

```
17 -> A.V
```

The pattern `add` also defines a do-part, which is executed when `add` is used as a procedure. The do-part of `A` may be executed directly by the following statement (the exact semantics should be clear when the complete mapping of the `Calculator` pattern is depicted below):

```
2 -> A -> X
```

To obtain the full semantics of the `add` pattern, it is mapped into the following inner class of `Calculator`:

```
class add extends Object {
   int V;
   void enter(int a) { V = a; }
   void do() { R = R + V; }
   int exit() { return R; }
}
```

In addition, the following method is added to class `Calculator`:

```
int add(int V) {
    add A = new add();
    A.enter(V);
    A.do();
    return A.exit();
}
```

The BETA invocation `5 -> C.add -> X` may now be mapped into the corresponding Java invocation `X = C.add(5)`.

The structure of the complete mapping of the `Calculator` pattern is:

```
class Calculator extends Object {
   int R;
   void set(int a) { ... };
   int add(int V) { ... }
   class set extends Object { ... }
   class add extends Object { ... }
```

```
}
```

As can be seen, each inner pattern of `Calculator` – in this case `set` and `add` – gives rise to a method and an inner class. This reflects the fact that a pattern may be used as a method as well as a class. The method `int add(int)` is for using pattern `add` as a method.

The inner class `add` may be used to create instances of the pattern as a class. Consider the BETA example of creating instances of pattern `add`. This is mapped to the following Java code.

```
Calculator C = new Calculator(); // C: @Calculator
Calculator.add A = C.new add();  // A: ^C.add; &C.add[]->A[]
A.V = 17;                        // 17 -> A.V
A.enter(2);                      // 2 -> A -> X;
A.do();
int X = A.exit();
```

The above example illustrates how a subset of BETA is mapped to Java. Of the remaining parts of BETA, the most challenging ones to map are `inner`, virtual patterns used as virtual classes, `leave` and `restart` (essential goto) out of nested method calls, patterns as variables and coroutines, concurrency and synchronization. The full mapping is described in [1]. [1] also describes the mapping to C#, which at the language level essentially is identical in terms of structure to the mapping to Java. One major difference is that .NET/CLR and thus C# does not support full inner classes, which means that the mapping of nested classes has to include an explicit reference to the enclosing object. At the bytecode level there are of course minor technical differences between JVM and CLR.

3.4 Using Java classes from BETA and vice versa

To import Java class files into a BETA program, BETA has been extended with an `ExternalClass` declaration that allows external classes to be imported. A tool has been implemented that can translate any class-file into a set of BETA external class declarations. In this way, the whole Java library is available from BETA.

Currently, the tool has to be used on a class-file before it can be used from BETA. In the future, it will be possible to import Java class files directly to a BETA program and integrate the tool with the compiler.

The other way around – using BETA from Java – works immediately. A BETA class-file can be imported and used from any Java program. JBeta generates a Java package structure from a BETA pattern. The main problem in using BETA from Java is the lack of a browser for BETA class files to be used by Java programmers. Currently, Java programmers will either have to inspect the BETA code and from that derive how the corresponding Java class files look like or alternatively reverse engineer a Java source file from the BETA class-file. However, it should be straightforward to develop a browser

tool.

4 JVM Based Eclipse Integration

In section 2, it was described how the indenter library, written in BETA, was accessed from Java in the plug-in using JNI. With JBeta this is much easier.

The Java class `BetaAutoIndentStrategy` can now be replaced with the BETA pattern specifying inheritance from the Java class `DefaultAutoIndentStrategy`:

```
BetaAutoIndentStrategy: DefaultAutoIndentStrategy
  (# customizeDocumentCommand:
       (# d: ^IDocument; c: ^DocumentCommand; value: @integer;
       enter (d[], c[])
       do ... 10 -> indentLine -> value (* Call BETA *)
       #)
  #)
```

This pattern calls the indentation library directly, so with an implementation of BETA on JVM, it is possible to drop all of the external libraries used for JNI. Note that the solution could just as well have kept `BetaAutoIndentStrategy` in Java, and accessed the BETA library from there. This is up to the programmer.

4.1 Debugging

The BETA IDE described so far gives the user the possibility to edit, indent, compile and run BETA programs. Another important part of an IDE is the debugger. As mentioned above, MjolnerTool has a native debugger, so there are two solutions for adding a debugger to the BETA plug-in: use the classic BETA debugger or somehow use the Java debugger available in the Eclipse Java-IDE, called the JDT debugger.

Using the classic BETA debugger would require the following: First, the existing code should be refactored, structuring it so there is a clear separation of UI and non-UI code. Second, the non-UI part of the debugger would then have to be modified making it fit in the Eclipse launching/debugging framework – which means it would have to fulfil the contracts of the plug-ins by implementing the correct Java interfaces.

Using the JDT debugger is possible because it operates on Java bytecodes and that is what JBeta generates. So the idea is to compile the BETA source using JBeta and when debugging, then do "whatever JDT does when debugging".

4.2 Implementation of the BETA Debugger in Eclipse

Launching and debugging in Eclipse, and thereby in JDT, is closely connected [14].

Since JBeta generates Java bytecodes from BETA sources the extension in the BETA IDE can use the same JDT class for handling, launching and debugging as is used for Java.

Using this class it is immediately possible to *launch* BETA applications. *Debugging*, however, is a bit more complicated, since Eclipse must be instructed on how to set the breakpoints, given the source BETA code.

The Eclipse class `JDIDebugModel`[17] [8] is an implementation of the Java Debug Interface (JDI) in SUN Microsystems' Java Platform Debugger Architecture (JPDA)[16]. It contains static methods for adding breakpoints to the debug model. The methods take arguments such as name of the source file, line number and the name of the type (class) in which to set the breakpoint. With this information the debugger is able to identify the corresponding bytecodes at run time.

What complicates the setting of breakpoints in BETA sources, is the *type name* that has to be specified. As described in section 3, a BETA source file is compiled into several `class` files, i.e. many types. Given a specific BETA file and a line number it is not trivial to find the corresponding generated `class` file. In a future version, the BETA compiler would, however, be able to supply this information for the BETA plug-in [9].

The Eclipse debugging model could be more friendly for BETA and the numerous other languages targeting JVM [15] by using a less strict coupling to JPDA.

Since Java class files contain information about source file names and line numbers for the bytecodes, the debugger should be able to find the break locations given just the source file name and line number: For classes already loaded, the debugger could check if any of these corresponds to the given source file, and set the breakpoint. If no such class is currently loaded, the debugger could defer the breakpoint setting and repeat the analysis whenever a new class is loaded.

In the proof-of-concept implementation of the BETA IDE, it is possible to add *line breakpoints* to the BETA source [10]. I.e. the debugger can stop execution on specific BETA imperatives. It is possible to perform single stepping, and objects may be inspected and modified with full source code information including BETA names of classes and variables [3]. This behaviour is mainly a property of JVM, which has complete type information, symbolic names, and source code line numbers for a given program. It is, however, impressive that it is possible to develop generic tools like Eclipse and JVM that given a compiler that generates JVM bytecode can support a full symbolic debugger

[8] located in package `org.eclipse.jdt.debug.core`

[9] The activation of JBeta when building BETA projects is accomplished by calling an external `jbeta` executable

[10] At the time of writing it is only possible to break inside the `PROGRAM` fragment, because we have not yet integrated the above mentioned program analysis to determine the class name from a source file position

for a considerably different language like BETA.

In figure 2 a simple BETA program is being debugged. The debugger has stopped the execution and it is possible to inspect and change the object fields.

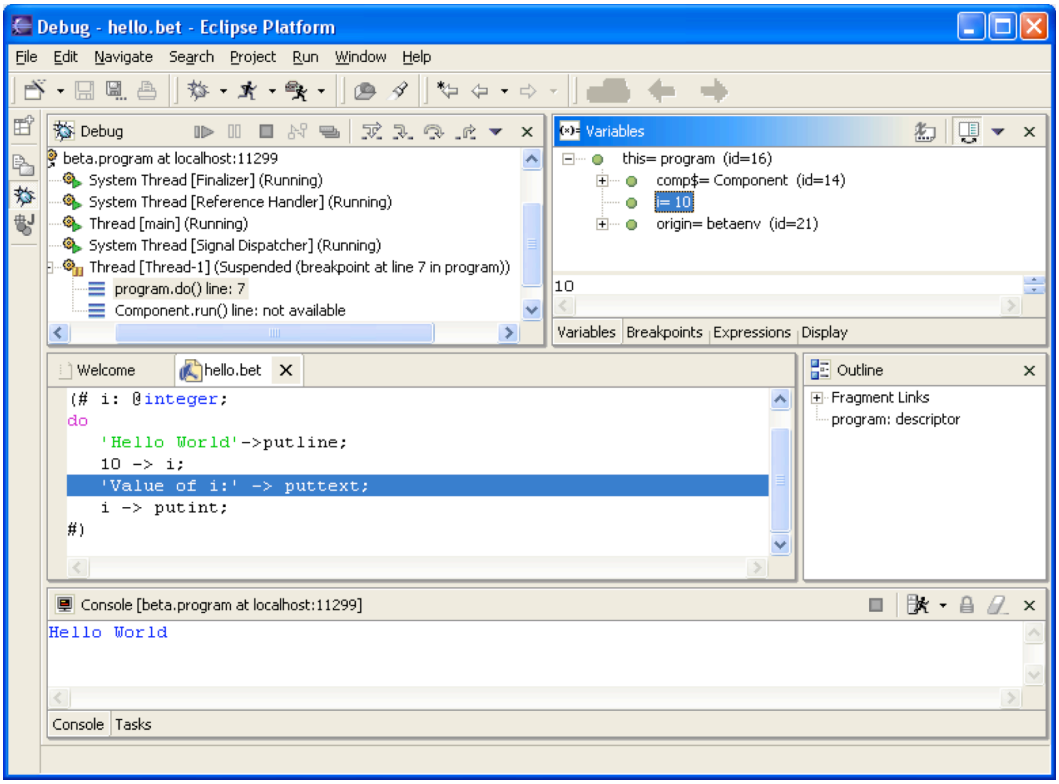

Fig. 2. Debugging BETA in Eclipse

4.3 Evaluation of the JBeta Integration

The Eclipse integration based on JVM is very elegant compared to the integration based on JNI. In JNI there were many layers of code, because calls between Java and BETA had to happen through C. With JBeta there is no difference in using BETA patterns or Java classes, so from the point-of-view of the BETA programmer, it is as if only working with BETA – when in fact it is the design of a plug-in that has to fit in the Eclipse framework. The language barriers have disappeared.

It is impressing that the JDT debugger in Eclipse is general enough to debug BETA at a symbolic level, despite originally being targeted at Java. Besides the relative small extension that has to be implemented to allow for the debugger to be launched, the only complex programming task is the handling of breakpoints.

The disadvantage of this solution is, as illustrated, that the mapping of the BETA language to JVM is non-trivial. Furthermore, a system like the BETA Mjølner system contains a large number of libraries and porting these involves

changing numerous system specific calls into calls of the corresponding Java class libraries.

4.4 Comparison with Microsoft Visual Studio .NET Integration

Part of the project[1] in porting BETA to CLR/.NET has been to investigate the use of Microsoft Visual Studio .NET [12] as an IDE supporting BETA. Visual Studio .NET (VS) is a powerful IDE supporting a number of languages and in this way, it is a possible alternative to Eclipse. As indicated by its name, this IDE supports the languages of the Microsoft .NET platform, which in many ways resembles the Java platform. Unlike Java, .NET is designed to support multiple programming languages.

VS immediately allows for browsing and editing of BETA source files. Furthermore, the multi-language support manifests itself by the ease of getting a BETA program into VS and doing almost complete source level debugging on it[11]. To make VS do this, all that was needed was to include source file and line number information into the generated byte code for .NET/CLR, and then just open an EXE file generated from BETA in VS. As can be seen from the VS screenshot in figure 3, this immediately allows VS to identify the source files, do source level stepping, insert break points, inspect objects and even change values of variables. No changes to VS were needed to obtain this!

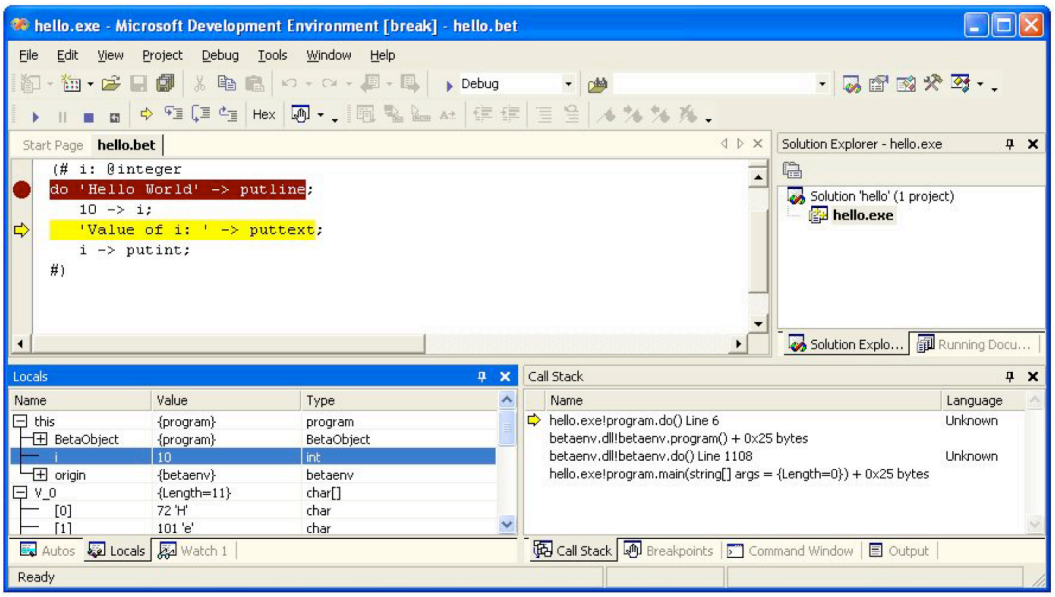

Fig. 3. Debugging BETA in Visual Studio .NET

However, to obtain a more complete language integration, i.e. syntax colouring, semantic tool-tips, control of the BETA compiler etc., hard work is needed:

[11] As can be seen in the VS screenshot in figure 3, variables are shown in internal .NET syntax, e.g. the `i` variable is shown to be of type `int`. Presenting this as the BETA type `integer` is one of the things that can be accomplished by extending the debugger integration

Like in Eclipse, the VS IDE is programmable and extensible. The APIs for doing this – called VSIP [12] – consist of a large number of COM interfaces, that need to be implemented to obtain the integration. COM[5] is Microsoft's predecessor to the .NET platform and it uses a completely different programming model and run-time system. Thus it requires a lot of non-.NET work to do the integration. However, at the time of writing, a beta-release of a so-called *managed VSIP API* has been announced from Microsoft [13]. Using this it will be possible to write the VS integration code entirely in BETA and compile this BETA code into .NET bytecode, i.e. using exactly the same strategy as is used for the Eclipse integration. This work has not been done, but the VSIP promises features such as handling of projects; participating in the building, debugging and deployment of a project; creating a custom debugger for your language; creating a custom text editor for your language; creating an interface builder with code generation in your language; IntelliSense (semantic tool-tips); syntax colouring; and adding your logo to the VS splash screen.

5 Conclusion

At this stage of the project, a proof-of-concept for integration of BETA in Eclipse has been obtained. Two strategies have been examined.

An integration based on JNI showing that it is indeed possible to write part of a plug-in – the indenter – in BETA and load it in Eclipse. There are, however, a number of disadvantages of this technique. JNI is not easy to use – this is more a Java problem than a problem of Eclipse – it should be possible to supply a much simpler and easy-to-use interface to C for Java than JNI. Even with an improved JNI, the *distance* from Java to BETA over C is too long, which severely complicates debugging and makes the code less readable.

The integration based on JVM on the other hand works well. By generating Java class files from BETA there are – except for the languages – no differences between writing plug-ins in Java or BETA.

In addition, the JVM integration makes it possible to use the Eclipse debugger for BETA at the same symbolic level as for Java. But there is room for improvement in the design of the JDT, e.g., breakpoint setting could be handled more generically – JDT is based on the JPDA model for the Java language, but could probably be changed to support other languages as well.

A corresponding integration into VS will be possible, and although the initial integration (browse, edit, and debug) requires less work than in Eclipse, to obtain a full language integration, comparable work is expected to be required if using the pre-released managed APIs, whereas more tedious work is expected if using the old style COM interfaces. One disadvantage of using VS is that it is a commercial product where Eclipse is free. In addition, VS only runs on Windows where Eclipse is available wherever Java is available.

[12] Short for VS Integration Programme

Our experience is, however, that Eclipse does not run well on certain Unix platforms including Sun Solaris.

It should be apparent that the main reason for the tight integrations of BETA with Eclipse and VS is that the integrations are based on the BETA implementations of BETA for JVM and CLR. The architecture of JVM and CLR have demonstrated that with modern virtual machines with full symbolic information available it is possible to obtain support for a new language like BETA.

The implementations of BETA on JVM/CLR have also demonstrated that language interoperability is possible between BETA and Java and C#. From BETA it is easy and straightforward to use Java and/or C# classes including inheritance. And the other way around – using BETA patterns from Java and/or C# works just as well. At this stage it is unknown to what extent BETA programmers will use Java/C# libraries and frameworks in major projects and vice versa. Currently, mainly example programs including Applets have been written.

It was no surprise that language interoperability worked for CLR/.NET, which has been designed for that. It was a surprise that it worked just as well for JVM.

It is outside the scope of this paper to discuss the quality of JVM and CLR with respect to supporting a language like BETA – see [1] for such a discussion. It was indeed possible to find a mapping of BETA to JVM/CLR. Certain parts of BETA did not map very well from a logical as well as an efficiency point of view. Platforms like JVM/CLR make it easy for language developers to write compilers. One might argue, though, that JVM and CLR are just representatives of the early generations of general purpose virtual machines. There is definitely room for improvements with regard to generality.

At this stage of the project, the main emphasis has been on the language interoperability issues in order to be able to write plug-ins in BETA. As a result writing Eclipse plug-ins is now possible in BETA and more sophisticated plug-ins can be created using the existing BETA libraries currently used by MjolnerTool.

The BETA IDE in Eclipse has been used by a small team of people for browsing, text editing, indentation and compilation. This has been a promising experience and in the future increased use of Eclipse as an IDE for BETA is to be expected. Furthermore, when a full integration into Visual Studio has been produced, using BETA in Visual Studio is to be expected as well. Time will tell which of the two will turn out to be the preferred IDE on Windows.

The main conclusion so far is that language interoperability as discussed in this paper is indeed possible to a certain extent. Modern virtual machines like JVM and CLR are able to support other languages including support for tools in IDE's as Eclipse and Visual Studio. Language interoperability should, however, be taken with a grain of salt. JVM and CLR support a family of languages with similar structure. The more a language differs from

the family, the more complicated is the mapping to the virtual machine. The authors do, however, believe that there is sufficient room between platforms like JVM/CLR and general platforms like standard microprocessors to support a broader range of languages at the same level as JVM and CLR does.

References

[1] P. Andersen, O.L. Madsen: *Implementing BETA on Java Virtual Machine and .NET – an exercise in language interoperability*, Department of Computer Science, University of Aarhus, January 2003.

[2] Eclipse Project Home Page: http://www.eclipse.org

[3] *M. B. Enevoldsen: Object Oriented Language Interoperability – A Case Study of BETA support in Eclipse*, Department of Computer Science, University of Aarhus, forthcoming Master Thesis, 2004.

[4] Arnold, Gosling, Holmes: *The Java Programming Language*, Third Edition, Addison-Wesley, Boston 2000, ISBN: 0201704331

[5] Dale Rogerson: *Inside COM*, Microsoft Press, February 1997 ISBN: 1572313498

[6] Microsoft .NET: http://www.microsoft.com/net

[7] Microsoft .NET Common Language Runtime: http://msdn.microsoft.com/library/en-us/cpguide/html/cpconCommonLanguageRuntimeOverview.asp

[8] C# home page: http://msdn.microsoft.com/vcsharp/language

[9] The Mjølner System, Mjølner Informatics A/S: http://www.mjolner.dk/mjolner-system

[10] O.L. Madsen, B. Møller-Pedersen: *Virtual Classes - A Powerful Mechanism in Object-Oriented Programming*, in: Proceedings of OOPSLA'89, Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, 1989

[11] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*. ACM Press/Addison Wesley, Addison-Wesley, June 1993, ISBN 0-201-62430-3, 350 pages. Out of print – a copy can be downloaded from http://www.mjolner.com/mjolner-system/books_en.php

[12] Microsoft Visual Studio Home Page: http://msdn.microsoft.com/vstudio

[13] Microsoft Visual Studio Industrial Partners (formerly: Microsoft Visual Studio Integration Program): http://www.vsipdev.com

[14] J. Szurszewski: *We Have Lift-off: The Launching Framework in Eclipse*, 2003 http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

[15] Programming Languages for the Java Virtual Machine: http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[16] Java Platform Debugger Architecture: http://java.sun.com/j2se/1.4.2/docs/guide/jpda

[17] Eclipse Online Documentation for JDIDebugModel: http://help.eclipse.org/help21/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/debug/core/JDIDebugModel.html

6 Appendix. A short overview of BETA

In BETA abstraction mechanisms such as class, type, procedure, method, function, coroutine, process and exception have been unified into one abstraction mechnism - the **pattern**. In addition to the pattern, BETA has subpattern, virtual pattern and pattern variable. This unification of abstraction mechanisms gives a uniform treatment of abstraction mechanisms and a number of new ones. Most object-oriented languages have classes, subclasses and virtual procedures and some have procedure variables. Since a pattern is a generalization of say class, procedure, function, the notions of subpattern, virtual pattern, nested pattern and pattern variable also apply to these abstraction mechanisms.

The **subpattern** covers subclasses as in most other languages. In addition, procedures may be organized in a subprocedure hierarchy in the same way as classes may be organized in a subclass hierarchy. Since patterns may also be used to describe functions, coroutines, concurrent processes, and exceptions, these may also be organized in a pattern hierarchy.

The notion of **virtual pattern** covers virtual procedures as in Simula, Java and C#. In addition, virtual patterns cover virtual classes, virtual coroutines, virtual concurrent processes, and virtual exceptions. Virtual classes provide a more general alternative to generic classes as in Eiffel or templates as in C++.

A **nested pattern** is a pattern defined textually within another pattern. Nested procedures are known from Algol and Pascal. In addition to nested procedure patterns, BETA also supports nesting of class patterns. Java supports nested classes in the form of inner classes. In C++ and C# a restricted form of nested classes is supported, but data-items in the enclosing object cannot be referenced from inner classes.

BETA includes the notion of **pattern variable**. This implies that patterns are first class values, that may be passed around as parameters to other patterns. By using pattern variables instead of virtual patterns, it is possible dynamically to change the behavior of an object after its generation. Pattern variables cover procedure variables (i.e. a variable that may be assigned different procedures). Since patterns may be used as classes, it is also possible to have variables that may be assigned different classes, etc.

An object may be **singular**, which means that it is described directly and

not as an instance of a class. In Java this corresponds to anonymous classes.

BETA supports **passive** as well as **active objects**. A BETA object may act as a coroutine. BETA coroutines may be executed concurrently (non pre-emptive scheduling in current version). The basic mechanism for synchronization is the semaphore, but high-level abstractions for synchronization and communication, hiding all details about semaphores, are easy to implement, and the standard libraries include monitors and Ada-like rendezvous. The user may easily define new concurrency abstractions including schedulers for processes. The distribution library supports true concurrency between BETA objects.

An example of a BETA program is given below:

```
(#
   Calculator:
      (# R: @integer
         set: (# V: @integer enter V do V -> R #);
         add: (# V: @integer enter V do R + V -> R exit R #)
      #);
    C: @Calculator;
    X: @integer;
do 12 -> C.set;
   5 -> C.add -> X
#)
```

- A BETA program is a singular object containing three declarations:
  - `Calculator`, a pattern; `C`, an instance of `Calculator`; and `X`, an instance of `integer`.

  and two statements:
  - `12 -> C.set`, a method invocation of `C.set` with argument 12.
  - `5 -> C.add -> X`, a method invocation of `C.add` with argument 5 and a subsequent assignment of the result to variable `X`.
- The `Calculator` pattern contains three declarations: `R`, an instance of integer; `set`, a pattern; and `add`, a pattern.
- The pattern `set` contains:
  - A declaration, `V`; an enter-part, `enter V`, which specifies that `V` is an input argument to `set`; and a statement `V -> R` assigning `V` to `R`.
- The pattern `add` is similar to `set`, but in addition contains:
  - An exit-part, `exit V`, which specifies that the value of `V` is returned as a result of execution of `set`.

In the above example, `Calculator`, `set` and `add` are all examples of patterns. `Calculator` is used as a class and `set` and `add` are used as methods. It is, however, possible to create instances of `set` and `add` as if they were classes. An example of creating instances of `add` is given in the beginning of section 3.3 of this paper.