On the Integration of Smalltalk and Java

Practical Experience with STX:LIBJAVA

Marcel Hlopko
Czech Technical University in Prague
marcel.hlopko@fit.cvut.cz

Jan Kurš
Software Composition Group, University of Bern
kurs@iam.unibe.ch

Jan Vraný
Czech Technical University in Prague, eXept Software AG
jan.vrany@fit.cvut.cz

Claus Gittinger
eXept Software AG
cg@exept.de

Abstract

After decades of development in programming languages and programming environments, Smalltalk is still one of few environments that provide advanced features and is still widely used in the industry. However, as Java became prevalent, the ability to call Java code from Smalltalk and vice versa becomes important. Traditional approaches to integrate the Java and Smalltalk languages are through low-level communication between separate Java and Smalltalk virtual machines. We are not aware of any attempt to execute and integrate the Java language directly in the Smalltalk environment. A direct integration allows for very tight and almost seamless integration of the languages and their objects within a single environment. Yet integration and language interoperability impose challenging issues related to method naming conventions, method overloading, exception handling and thread-locking mechanisms.

In this paper we describe ways to overcome these challenges and to integrate Java into the Smalltalk environment. Using techniques described in this paper, the programmer can call Java code from Smalltalk using standard Smalltalk idioms while the semantics of each language remains preserved. We present STX:LIBJAVA — an implementation of Java virtual machine within Smalltalk/X — as a validation of our approach.

IWST '12 August 28, Gent.
Copyright © 2012 ACM 978-1-4503-1897-6/12/08...$15.00

Categories and Subject Descriptors H.3.3 [*Programming Languages*]: Language Constructs and Features—Language Interoperability; H.3.4 [*Programming Languages*]: Processors—Interpreters, Virtual Machines

General Terms Language Interoperability

Keywords Language Interoperability, Smalltalk, Java

1. Introduction

Without doubt, the Java programming language has become one of the most widely used programming languages today. A significant amount of code is written in Java, ranging from small libraries to large-scale application servers and business applications. Nevertheless, Smalltalk still provides a number of unique features (such as advanced reflection support or expressive exception mechanism) lacking in Java, which makes Smalltalk suitable for many kinds of project. It is a tempting idea to call Java from Smalltalk and vice versa, as it enables the use of many Java libraries within Smalltalk projects.

The idea of Java-Smalltalk integration is not new and has been explored by others in the past. JavaConnect (Brichau and De Roover [1]) and JNIPort (Geidel [4]) use foreign function interfaces to connect to the Java virtual machine and to call a Java code. Bridges, such as VisualAge for Java (Deupree and Weitzel [2]) or Expecco Java Interface Library (Expecco [3]), use proxy objects, which intercept and forward function calls and return result values or handles via an interprocess communication channel. Another more versatile approach, is to execute both languages within the same virtual machine and use a common object representation for both. STX:LIBJAVA, which is presented in this paper, is an example of such an approach: it executes Java within the Smalltalk virtual machine. Another example is Redline

Smalltalk (Ladd [9]), which executes Smalltalk using standard Java virtual machine.

Seamless, easy to use integration of two programming languages consists of various parts. First, it must allow one language to call functions in the other, possibly passing argument objects and getting return values (*runtime-level* integration).

Second, it should support programmer-friendly argument and return value passing between the languages.

Third, it should ideally preserve object identity. The use of replicas or proxy objects can introduce various problems when objects are stored or managed by their identity. This also affects any side effects to such objects when calling functions in the other language.

Finally, it should seamlessly integrate the two languages on the syntactic level, which means that (ideally) no additional glue or marshalling code should be required to call the other language (*language-level* integration).

In the case of Java and Smalltalk, language-level integration raises a number of challenges, due to their different design and semantics. In particular these are:

- Smalltalk uses keyword message selectors, whereas Java uses traditional C-like selectors (virtual function names).
- Java supports method overloading based on static types, whereas there is no static type information in Smalltalk.
- Exception and locking mechanisms differ.

In this paper, we present STX:LIBJAVA, a Java virtual machine (JVM) implementation built into the Smalltalk/X environment. STX:LIBJAVA allows for the program to call Java code from Smalltalk almost as naturally as normal Smalltalk code. We will demonstrate how STX:LIBJAVA integrates Java into Smalltalk and we will describe how it deals with semantic differences.

The contributions of this paper are (i) a new approach of Smalltalk and Java integration, (ii) identification of problems imposed by such an integration and (iii) solutions for these problems and their practical validation in STX:LIBJAVA.

The paper is organized as follows: Section 2 discusses integration problems in detail. Section 3 gives an overview of STX:LIBJAVA and its implementation. Sections 5, 4, 6 and 7 describe techniques to solve these problems. Section 8 presents some real-world examples to validate our solution. Section 9 discusses related work. Finally, Section 10 concludes this paper.

2. Problem Description

At first glance Smalltalk and Java are very similar. Both are high-level object oriented, class-based languages with single inheritance and automatic memory management. In both languages message sending is the fundamental way of communication between objects. In this section we enumerate details by which these languages differ and which pose problems when integrating these languages into a common system.

1. **Class access.** In Smalltalk classes are identified by name. There is only one class with a given name at any time (although it may change over time) and it must be present and resolved prior to the code actually beeing executed. The system triggers a runtime error otherwise. The process of loading classes into the system is not specified in the standard, although some Smalltalk dialects provide namespaces and/or a lazy class loading facility.

   On the other hand Java provides a well-defined, user-extensible mechanism called *classloaders* for lazy-loading of classes into a running system. Moreover in Java a class is not only identified by its name but by its defining classloader as well. In other words two possibly different classes with the same name may coexist in the running system as long as they have been defined by different classloaders. Which classloader is used to load a particular class at a particular place in the code is a subject to complex rules and depends on the runtime context.

2. **Selector mismatch.** On the bytecode level, a method is identified by a *selector* in both languages. However, the syntactic format of selectors differs. Smalltalk uses the same selector in bytecode and in a source code. Java encodes type information into a selector at the bytecode level. This may affect reflection and/or dynamic execution, eg via `#perform:`.

   For example, consider the following code in Smalltalk:

   ```
   out println: 10
   ```

   Let's assume that `out` is an instance of the Java class `java.io.PrintStream`. Smalltalk compiles a message send with the selector `println:`. However, at the bytecode level the Java selector is `println(I)V`, whereas `println` is the method name expressed in the source code. The added suffix `(I)V` means that the method takes one argument of the Java type `int` and `V` means that the method does not return a value (aka returns a void type).

3. **Method overloading.** Java has a concept of overloading: in a single class, multiple methods with same name but with different numbers or types of arguments may coexist. For example, a `PrintStream` instance from the previous example has multiple methods by the name `println`, one taking an argument of type `int`, another taking a Java `String` argument, and so on. On the virtual machine level, each overloaded method has a different selector (`println(I)V`, respectively `println (Ljava/lang/String;)V`).

   The method which is called depends on the static types of the argument types at the call site. At compile time the Java compiler finds the best match and generates a send with the particular selector. For example, the source code

```
out.println("String");
out.println(1);
```

produces the following bytecode with type information encoded in method selectors (arguments of `INVOKEVIRT` instruction):

```
ALOAD ...
LDC 1
INVOKEVIRT println(L...String;)V
BIPUSH 1
INVOKEVIRT println(I)V
```

As Smalltalk is dynamically typed, no type specific selector are chosen by the Smalltalk compiler at compile time.

4. **Protocol mismatch.** Comparing Java and Smalltalk side by side, many classes with similar functionality are found. For example both Java's `java.lang.String` and Smalltalk's `String` class represent a `String` type. More complex examples are `java.util.Map` and Smalltalk's `Dictionary` class. Here, the classes have different names, possibly have different method names, but their purpose and usage is similar — both store objects under a particular key.

   When using code from both languages the programmer has to take care which kind of object (Java or Smalltalk) is passed as an argument. For example the hash of a string is obtained by sending the `hashCode` message in Java but `hash` in Smalltalk. Passing a Smalltalk String as an argument to a Java method which expects a Java-like String may result in a `MethodNotFound` exception.

5. **Exceptions.** The exception mechanisms in Smalltalk and Java differ in two very profound ways:

   First in Smalltalk both exception and ensure-handlers are blocks. Blocks are self-contained first-class closures that can be manipulated and executed separately. In Java exception and finally-handlers are syntactic entities and technically generate a sequence of bytecode instructions, which are spliced into the instruction stream. They can be only executed by jumping to the beginning of the handler and then continuing execution from that location. The method in question contains a special *exception table* which maps instruction ranges to the beginning of a particular handler.

   Second, Smalltalk provides resumable exceptions. When an exception is thrown (raised) in Smalltalk, the handler is executed on top of the context that caused that exception. The underlying stack frames are still present at handler execution time. If the handler returns, *i.e.,* the exception is not "proceeded"[1], ensure blocks are executed after the exception handler. Java provides only return semantics. When a handler is found, all contexts up to the handler context are immediately unwound and the execution is resumed at the beginning of the handler. Any finally block is treated like special exception handler, which matches any exception.

   In other words, the main difference between Smalltalk and Java exceptions from the programmer's point of view is that in Smalltalk, ensure blocks are *eventually* executed *after* the the handler, if the handler decides to return. In contrast, finally blocks of Java are *always* executed and their evaluation happens *before* the execution of the handler. A Smalltalk handler is even free to dynamically decide whether to proceed with execution after the exception is raised. Such proceedable exceptions are useful to continue execution after fixing a problem in the handler, or to implement queries or notifications (for example, to implement loggers or user interaction).

6. **Synchronization.** The mechanisms for process synchronisation differ both in design and implementation.

   The principal synchronisation mechanism in Java is a *monitor*. Conceptionally every Java object has associated with it a monitor object. Synchronization is done using two basic operations: *entering* the monitor, which may imply a wait on its availablility and *leaving* the monitor (Section 8.13, Lindholm and Yellin [10]). Monitor *enter* and *leave* operations must be properly nested. In Java, whole methods can be marked as *synchronized*, in which case the Java virtual machine itself is responsible for *entering* the monitor associated with the receiver and *leaving* it when the method returns. This happens both for normal and unexpected returns, for example due to unwinding after an uncaught exception. For *synchronized blocks*, the compiler emits `MONITORENTER` and `MONITOREXIT` bytecodes and ensures that no monitor remains entered, no matter how the method returns. Again, both normal and unwinding (Section 8.4.3.6, Gosling et al. [8], Section 3.11.11, Lindholm and Yellin [10]). The compiler achieves this by generating special finally-handlers, which *leave* the monitor and rethrow the exception.

   In Smalltalk (Smalltalk/X, in particular), processes are usually synchronized by semaphores. The programmer is responsible for proper semaphore signaling, although library routines provide support for critical regions. Technically speaking, there is no support at the virtual machine level for semaphores, except for a few low-level primitive methods.

   Now consider the following code:

```
[
  self basicExecute
] on: ExecutionError do: [ :ex |
```

[1] Smalltalk allows for exceptions to be "proceeded", which effectively means that the executions is resumed at the point where the exception was thrown (raised).

```
  self handleError: ex
]
```

A system which integrates Java and Smalltalk must ensure that all monitors possibly entered during execution of the `basicExecute` method are left when an `ExecutionError` is thrown and caught by the handler.

3. STX:LIBJAVA

3.1 In a Nutshell

STX:LIBJAVA is an implementation of the Java virtual machine built into the Smalltalk/X environment. In addition to providing the infrastructure to load and execute Java code, it also integrates Java into the Smalltalk development environment, including browsers, debugger and other tools.

Calling Java from Smalltalk is almost natural. Listing 1 demonstrates how a Java library can be used from Smalltalk. In the example, an XML file is parsed and parsed data is printed on a system transcript. The XML file is parsed by the SAX parser, which is completely implemented in Java. However, the SAX events are processed by a Smalltalk object — an instance of CDDatabaseHandler (see Listing 2).

This example demonstrates STX:LIBJAVA interoperability features:

- Java classes are referred to via a sequence of (unary) messages, which comprise the fully qualified name of a class. Java classes can be reached via the global variable `JAVA`. For example, to access `java.io.File` from Smalltalk, one may use[2]:

  ```
  JAVA java io File
  ```

- To overcome selector mismatch errors, STX:LIBJAVA intercepts the first message send and automatically creates a *dynamic proxy method*. These dynamic proxy methods translate the selector from Smalltalk keyword form into a correctly typed Java method descriptor and pass control to the corresponding Java method.
- STX:LIBJAVA provides a bridge between Java and Smalltalk exception models. Java exceptions can be handled by Smalltalk code. When a Smalltalk exception is thrown and the handler returns, all Java finally blocks between the raising context and handler context are correctly executed and all possibly entered monitors are left. No stale Java locks are left behind.

3.2 Architecture of STX:LIBJAVA

In this section we will briefly outline STX:LIBJAVA's internal architecture.

Unlike other projects which integrate Java with other languages, STX:LIBJAVA does not use the original JVM in parallel with the host virtual machine, nor does it translate Java source code or Java bytecode to any other host language. Instead the Smalltalk/X virtual machine is extended to support multiple bytecode sets and execute Java bytecode directly. To our knowledge, Smalltalk/X and STX:LIBJAVA is the only programming environment that took this approach.

The required infrastructure for loading `.class` (chapter 4, Lindholm and Yellin [10]) files, class loader support and additional support for execution, such as native methods, is implemented in Smalltalk. Java runtime classes and methods are implemented as customized Smalltalk `Behavior` and `Method` objects. In particular, Java methods are represented as instances of subclasses of the Smalltalk *Method* class. However, they refer to Java instead of Smalltalk bytecode. Execution of Java bytecode is implemented in the virtual machine. In the same way that Smalltalk bytecode is handled by the VM, Java bytecode is interpreted and/or dynamically compiled to machine code (jitted).

However, some complex instructions (such as `CHECKCAST` or `MONITORENTER`) are handled by the virtual machine calling back into the Smalltalk layer via a so-called trampoline and are implemented in Smalltalk as a library method. Similarly, all native methods are implemented in Smalltalk.

Both Smalltalk and Java objects live in the same object memory and are handled by the same object engine and garbage collector. Performance-wise, there is no difference between Smalltalk code calling a Java method or other Smalltalk code. Moreover, all dynamic features of the Smalltalk environment - such as stack reification and advanced reflection — can be used on the Java code.

The main disadvantage of our approach (as opposed to having a separate original JVM execute Java bytecodes) is that the whole functionality of the Java virtual machine. has to be reimplemented. This includes an extensive number of native methods, which indeed involve a lot of engineering work. However, we believe that this solution opens possibilities to a much tighter integration which would not be possible otherwise.

4. Class Access

In Smalltalk classes are stored by name in the global Smalltalk dictionary. Obviously, this dictionary cannot be reused by the Java subsystem, as Java classes are specified by name and defining class loader. Therefore loaded classes are accessed through `JavaVM` class. Listing 8 shows its usage in case of a known class loader instance.

```
JavaVM
  classForName: 'org.junit.TestCase'
  definedBy: classLoaderObject
```

Listing 3. Accessing Java class with known class loader

[2] alternatively, the class can also be referred to via a sub-namespace, as `JAVA::java::io::File`.

```
factory := JAVA javax xml parsers SAXParserFactory newInstance.
parser := factory newSAXParser getXMLReader.
parser setContentHandler: JavaExamples::CDDatabaseHandler new.
[
  parser parse: 'cd.xml'.
] on: JAVA java io IOException do:[:ioe|
  Transcript showCR: 'I/O␣error:␣', ioe getMessage.
  ioe printStackTrace
] on: UserNotification do:[:un|
  Transcript showCR: un messageText.
  un proceed.
]
```

Listing 1. Smalltalk code calling Java XML parser

```
CDDatabaseHandler>>startElement:namespace localName:localName qName:qName attributes: attributes
  tag := qName.

CDDatabaseHandler>>endElement:namespace localName:localName qName:qName
  qName = 'cd' ifTrue:[
    title isNil ifTrue:[self error: 'No␣title'].
    artist isNil ifTrue:[self error: 'No␣artist'].
    index := index + 1.
    UserNotification notify:
      (index printString , '.␣', title , '␣-␣' , artist)
  ]

CDDatabaseHandler>>characters: string offset: off length: len
  tag = 'title'  ifTrue:[
    title := string copyFrom: off + 1 to: off + len.
    tag := nil.
  ].
  tag = 'artist' ifTrue:[
    artist := string copyFrom: off + 1 to: off + len.
    tag := nil.
  ].
```

Listing 2. An excerpt of `CDDatabaseParser` used in Listing 1

As already shown in Listing 1, the `JAVA` global variable is provided to refer to a Java class using the current class loader.

Interoperability approaches based on foreign-function interfaces of the JVM and host virtual machine suffer from the inability to reclaim classes which have entered the JNI (Java native interface). In STX:LIBJAVA all loaded classes are reclaimed in compliance with the JVM specification (Section 12.7, Gosling et al. [8]).

5. Dynamic Proxy Methods

The *Dynamic Proxy Method* is a mechanism employed by STX:LIBJAVA to solve selector and protocol mismatch and to deal with Java's method overloading. A proxy method is an intermediate method that possibly performs an additional method resolution, transforms arguments and finally passes control to a real method dispatching on the type and number of arguments. Such a proxy is generated dynamically whenever the control flow crosses the language boundary, *i.e.,* when Smalltalk calls Java or vice versa. In the following sections we describe in detail how dynamic method proxies solve the problem outlined in the previous sections. We will demonstrate proxies on examples of Smalltalk calling Java; the actions performed in the opposite call direction are analogous.

5.1 Selector Mismatch

Consider the example given in the listing 3. Without additional interoperability support, a `DoesNotUnderstand` exception would be raised, since there is obviously no method for the `println:` selector in the Java `PrintStream` class.

```
out := JAVA java lang System out.
out println: 10.
```

Listing 4. Example of selector mismatch

STX:LIBJAVA's interoperability mechanism catches cross-language message sends and dynamically generates a *dynamic proxy method* for the original selector, which performs a second send using a transformed selector. The code of the proxy is shown on Listing 4. The proxy is compiled on the fly and installed into the receiver's class and the original message-send is restarted. This way a proxy method is generated only for the very first time and subsequent sends will use the "fast path", invoking the already generated proxy directly. Details on how sends are intercepted are discussed below in section 4.6.

```
java.io.PrintStream>>println: arg
  self perform: #'println(I)V'
          with: arg
```

Listing 5. A proxy method for `println()` Java method

5.2 Method Resolution

There are numerous ways to translate Smalltalk selectors to corresponding Java selectors and vice versa. This section does not discuss any pros and cons of possible approaches. We do not believe that there is the only one the best way how to translate selectors. STX:LIBJAVA simply uses the way which showed to be the most natural and easy for our purpose. The translation rules are described below.

Smalltalk to Java resolution. When calling a selector like `println:`, and the receiver is a Java object, the Java object and its super-classes are searched for all methods with a name of `println`. In case there is no such method, the `#doesNotUnderstand:` message is sent, as usual. If exactly one method exists by that name, that method is invoked. In case more than one method exists by the name (*e.g.,* the method is overloaded), the algorithm consults at run-time the number and types of arguments and tries to find the best matching Java method. If the number of arguments does not match, the `#doesNotUnderstand:` message is sent. Finally, if an interface type is expected as an argument and the argument is a Smalltalk object (which usually does not implement the Java interface), STX:LIBJAVA follows the traditional Smalltalk duck-typing philosophy and passes the Smalltalk argument unchanged to the method. Either the argument object implements any required interface methods (and everything works as expected then), or Java throws a runtime exception, which can be handled either by Smalltalk or by Java code.

Java to Smalltalk resolution. When calling a selector like `put(Ljava.lang.String;Ljava.lang.Object)V`[3]. and the receiver is a Smalltalk object, the object's class and its superclasses are searched for any selector starting with the first keyword part, `put:`. The rest of the selector is ignored in this matching process. However, the number of arguments must match. Also argument types are ignored. If more than one method fulfils these criteria, an `AmbiguousMessageSend` error is raised.

Variable number of arguments. Starting with Java 1.6, Java supports a variable number of arguments (section 8.4.1, Gosling et al. [8]). Technically, variable arguments are wrapped into an array object and passed to the method as a single argument. A Java compiler is responsible for generating code that wraps variable arguments. Therefore, no special care is required during method resolution.

5.3 Method Overloading

To demonstrate method overloading, we extend the example in Listing 3 , as depicted in Listing 5. After execution of line 3, a new proxy method has been added to the `java.io.PrintStream` class as depicted in Listing 4. The execution of line 4 would raise a runtime error, since we call the `println(I)V` method with a `boolean` parameter.

```
1 out := JAVA java lang System out.
2 out println: 10.
3 out println: true.
```

Listing 6. An example of overloaded method called from Smalltalk

```
java.io.PrintStream>>println: a1
  | method |
  (a1 class == SmallInteger) ifTrue:[
    ↑ self perform: #'println(I)V' with: a1
  ].
  self recompile: a1.
  ↑ self println: a1.
```

Listing 7. A proxy method for an overloaded method

Due to the dynamic nature of Smalltalk, argument types cannot be statically inferred and may even change during execution. Therefore, another method resolution step has to be added to the proxy method. To ensure type-safety (as

[3] A Java selector for method named `put` that takes two arguments of types `java.lang.String` and `java.lang.Object` respectively and whose return type is `void`

required and assumed by the Java code) the actual call to the Java method is protected by a “guard” that checks the actual argument types. The code for the `println:` proxy after execution of line 2 is shown in Listing 6.

Only one guard is added at a time. Line 6 ensures that if no guard matches — like when line 3 of example from Listing 5 is executed — the proxy is recompiled, possibly adding a new guard. Line 7 restarts the send. This prevents unnecessary guards which are actually never used from being generated. An alternative implementation based on multiple dispatch could be implemented by dynamically installing double dispatch methods in the encountered argument classes. However, as the number of dynamically encountered argument types is usually relatively small, we believe that the switch code on the argument class is usually sufficient and faster.

5.4 Protocol Mismatch

In some cases, it is not sufficient to simply translate the Java selector to a Smalltalk selector and vice versa. For example, Smalltalk code can send a `#isEmpty` message to a `java.lang.String` (because it contains the `isEmpty()` method), but it cannot send `#collectAll:`, as there is no such functionality in a Java string. Therefore the arguments and return values should also be converted when crossing the language boundary. STX:LIBJAVA makes these conversions automatically for predefined types (such as String, Integer, Boolean, ...). Thanks to dynamic proxy methods, user-defined types can be converted automatically as well.

Being aware of the protocol mismatch problem, we have to update the proxy method from Listing 6 as depicted in Listing 7. The `#asJavaObject` and `#asSmalltalkObject` are responsible for conversion between Smalltalk and Java types.

```
java.io.PrintStream>>println: a1
  |  jA1 jA1Class |
  jA1 := a1 asJavaObject.
  jA1Class :=
    Java classForName:
      'java/Lang/String'.

  (jA1 class == jA1Class) ifTrue: [
    ↑ (self #'println(Ljava/lang/String;)
        V': jA1)
          asSmalltalkObject.
  ].
  self recompile: a1.
  ↑ self println: a1.
```

Listing 8. A proxy method with argument and return value conversions

5.5 Field Accessing

In Java, public fields can be accessed directly using the dot-notation whereas in Smalltalk, values of instance variables could only be accessed using accessor methods. Although in modern Java, declaring instance fields `public` and accessing them directly is considered as a “bad style”, public static fields are often thorough Java libraries to expose constant values. STX:LIBJAVA allows for public field to be accessed from Smalltalk in a Smalltalk way,*i.e.,* by access methods. These accessor methods are dynamically compiled whenever needed - just like proxy methods described above.

Consider the example given in the listing **??** that access public static field `PI` of class `java.lang.Math`. STX:LIBJAVA interoperability mechanism catches the send of message `#PI` to the class and automatically generates getter method returning corresponding field.

```
cf = 2 * (JAVA java lang Math PI) * r
```

Listing 9. An example of accessing Java fields from Smalltalk

Accessor methods are generated only for Java fields declared as `public`. If the field is declared as `final`, only getter method is generated. Same mechanism is used to access both static and instance fields.

5.6 Intercepting the Message Send

To install a proxy, a message send must be intercepted. A standard Smalltalk solution would be to override the `#doesNotUnderstand:` method so it creates and installs the generated proxy method.

However, STX:LIBJAVA utilizes the method lookup metaobject protocol (MOP; Vraný et al. [12]) which is integrated into the Smalltalk/X virtual machine. The MOP allows for a user-defined method lookup routine to be specified on a per-class basis. This user-defined method lookup routine installs the proxy and invokes it.

6. Mixed Exception-Handling

Considering the mixed Smalltalk and Java code in Figure 1 (part a) which creates a user account, let us follow the actions taken when an exception is thrown in an invocation of `CreateAccountCmd»perform` (part b). First, the handler is searched and executed (Step 1 in Figure 1). Since the handler does not proceed the stack should be unwound and control passed to the `createAccountClicked` method. All ensure and finally blocks between the current context and the context for `createAccountClicked` method are executed first. The first such block is the ensure block in the `execute` method (Step 2 in Figure 1). The second is the finally block in `createAccount()`. However, the finally block is not a Smalltalk block and thus cannot be evaluated easily. Due to the underlying virtual machine implementation, the only way to execute the finally code is to (i) set the program counter to the beginning of the handler and (ii) restart the method. Restarting a method also implies that contexts below the restarted method's context are destroyed (*i.e.,* contexts for called methods). In particular, the context of the `raise` method should be destroyed. This poses a problem, as it is the context of the method which controls all handler and ensure block evaluation.

To solve this problem we exploit two facts. First, when ensure or finally handlers are executed, the contexts below the handler context are going to be destroyed anyway. Second, finally blocks are compiled in such a way that they never touch the exception object. The exception is only temporarily stored and then rethrown by the `ATHROW` Java instruction. The exception object can be any object, not necessarily a Java object inheriting from `Throwable`. To execute the finally code, we pass a special *finally token* object as an exception and restart the method with the finally block (Step 3 in Figure 1). This special *finally token* is recognised by the `ATHROW` instruction. Upon detection of that special token, `ATHROW` continues to evaluate finally and ensure blocks and finally unwinds the stack (Step 4 in Figure 1).

7. Synchronization

The Java runtime library has been designed to be thread-safe. Critical sections guarded by monitors are pervasive through the code. Correct handling of monitors in case of mixed Java-Smalltalk code is essential as a single leftover monitor can easily result in blocking the whole system.

Prior to entering a critical section, a monitor is *entered*, which must be *left* at the end of the section. This is done either implicitly by the virtual machine (when the whole method is marked as *synchronized*) or explicitly using special `MONITORENTER` and `MONITORLEAVE` instructions. Those implement fine-grain synchronization of Java's *synchronized blocks*.

7.1 Synchronized blocks

When an exception occurs during the execution of a synchronized block, the guarding monitor must be left before the control flow is passed to a handler higher up on an execution stack [4]. To ensure this, a Java compiler must generate a synthetic finally block in which the monitor is left and the exception is rethrown (Section 7.13, Lindholm and Yellin [10]). Figure 2 shows an example of such a finally block.

Consider the code shown in Figure 2. Because an exception could be thrown in the synchronized block (bytecode lines 2 - 10), the compiler generates a special handler (bytecode lines 11 - 15, Figure 2), in which the monitor is left and the exception is rethrown.

7.2 Synchronized methods

In the case of synchronized methods, neither `MONITOR-ENTER` / `MONITOREXIT` instructions, nor synthetic finally blocks are generated by the Java compiler. Instead, synchronization is done directly by the virtual machine. The monitor used for synchronization is the monitor associated with the receiver or (in case of a static method) with the class.

Prior to executing a synchronized method, the virtual machine *enters* the monitor associated with the receiver and tags the context as an *unwind context*. The VM intercepts returns through such tagged contexts and trampolines into the Smalltalk level for any unwind actions to be executed. In this case, the monitor-leave semantic is performed.

8. STX:LIBJAVA at Work

As mentioned in the 1 section, language integration consists of runtime-level and language-level integration.

Runtime-level integration. Several reasonably large Java projects have been chosen as benchmarks to validate the correct implementation of Java runtime support. These projects include:

- Apache Tomcat[5] – a Servlet/JSP container,
- SAXON[6] – an XSLT and XQuery processor,
- Groovy[7] – a dynamic language for Java platform,

Apache Tomcat makes heavy use of almost all Java features, from threads and synchronization, through dynamic class generation, class loading and finalization, to exceptions and finally blocks. Groovy makes heavy use of Java reflection and especially class loading as it dynamically generates Java bytecode and loads it into a running system. All these programs run correctly under STX:LIBJAVA.

Language-level integration. Language-level interoperability was already demonstrated in Listings 1 and 2, which use the Xerces XML parser to parse an XML file. The filename is passed into the Java method as a Smalltalk string. The SAX handler which is passed to the parser and later called for decoded elements is actually implemented in Smalltalk. No boilerplate code is needed whatsoever.

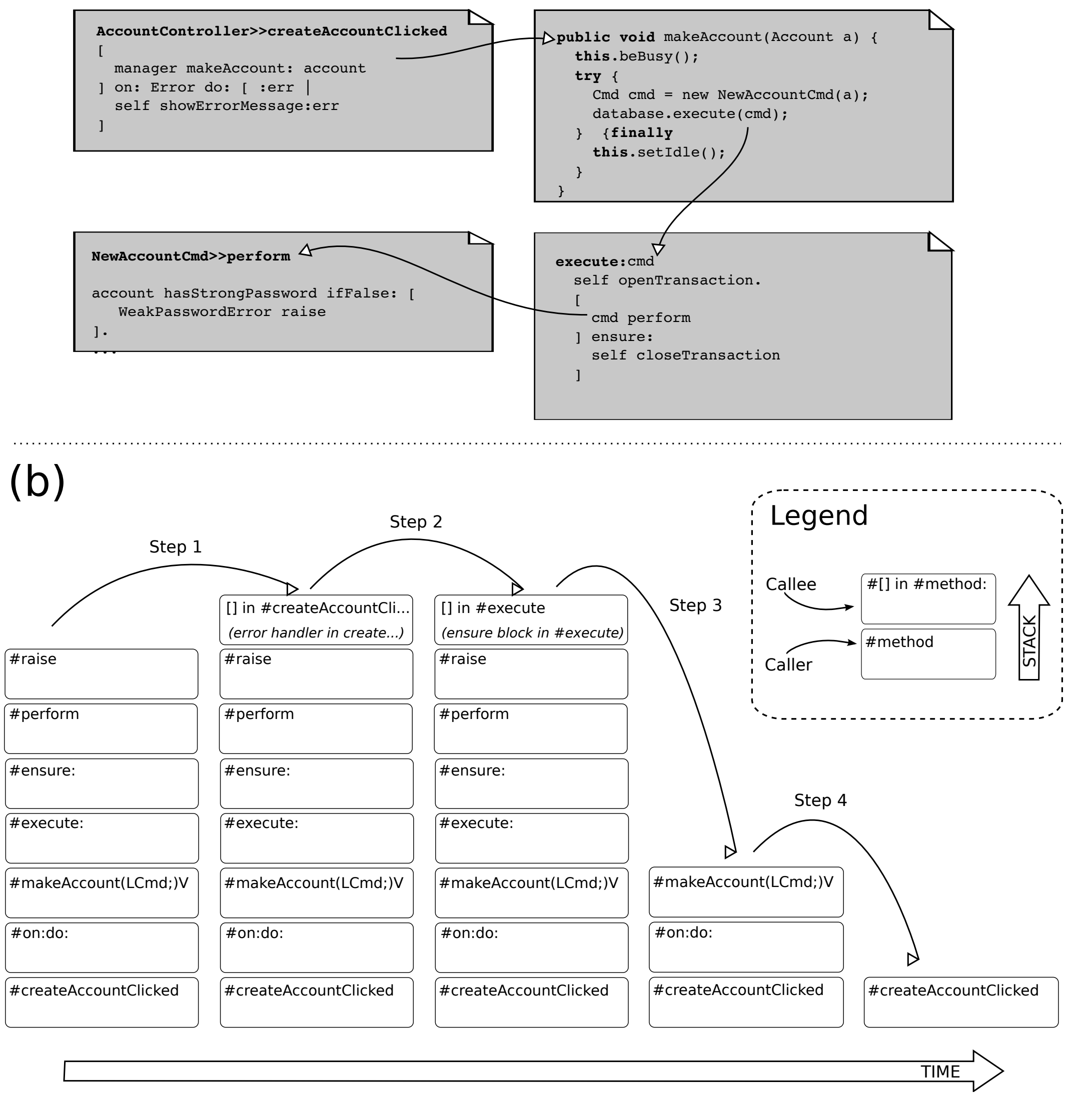
Tool support. A Java implementation in Smalltalk would not be complete without support in the development tools. Java classes can be browsed using the standard Smalltalk class browser, and Java objects or classes can be inspected in the Smalltalk inspectors. For programmer convenience, specialized inspectors are provided for specific Java classes such as `Vector`, `ArrayList`, `Set` or `Map`. Java is also fully supported by the debugger — breakpoints may be set on methods and Java code can be single stepped and debugged just like Smalltalk. A Groovy interpreter has been integrated into the Workspace application so programmers

[4] assuming that the stack grows downwards

[5] http://tomcat.apache.org

[6] http://saxon.sourceforge.com

[7] http://groovy.codehaus.com



```
AccountController>>createAccountClicked
[
  manager makeAccount: account
] on: Error do: [ :err |
  self showErrorMessage:err
]
```

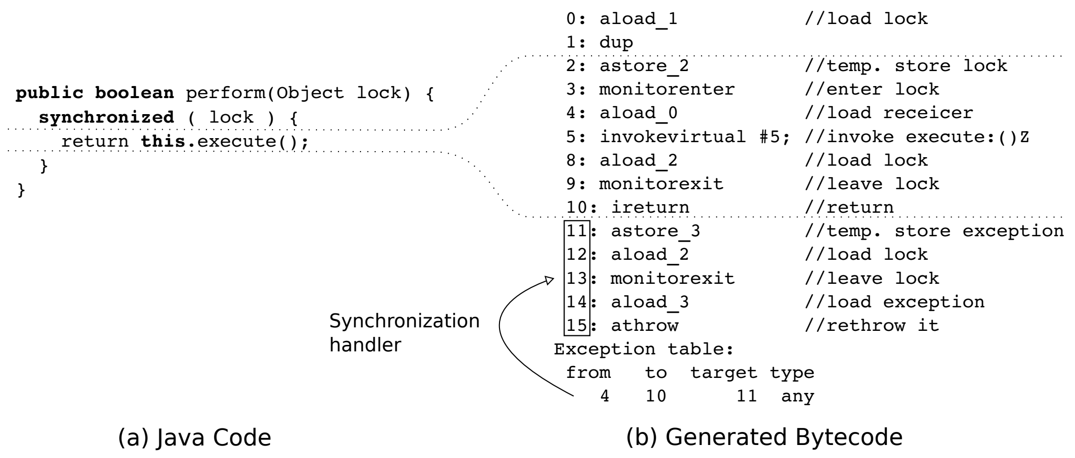
```
public void makeAccount(Account a) {
  this.beBusy();
  try {
    Cmd cmd = new NewAccountCmd(a);
    database.execute(cmd);
  }  {finally
    this.setIdle();
  }
}
```

```
NewAccountCmd>>perform

account hasStrongPassword ifFalse: [
   WeakPasswordError raise
].
...
```

```
execute:cmd
  self openTransaction.
  [
    cmd perform
  ] ensure:
    self closeTransaction
  ]
```

Figure 1. (a) an example of mixed-exception code (b) method activation stackswhen executing code from (a). Too keep the figure concise, unimportant intermediate contexts for "try" blocks in `#on:error:` and `#ensure:` are omitted.



```
public boolean perform(Object lock) {
  synchronized ( lock ) {
    return this.execute();
  }
}
```

```
 0: aload_1            //load lock
 1: dup
 2: astore_2           //temp. store lock
 3: monitorenter       //enter lock
 4: aload_0            //load receicer
 5: invokevirtual #5;  //invoke execute:()Z
 8: aload_2            //load lock
 9: monitorexit        //leave lock
10: ireturn            //return
11: astore_3           //temp. store exception
12: aload_2            //load lock
13: monitorexit        //leave lock
14: aload_3            //load exception
15: athrow             //rethrow it
Exception table:
 from   to  target type
   4    10      11  any
```

Figure 2. Example of synchronized block in Java

can quickly test Java code (doIt), in just the way they are used to with Smalltalk. Also, JUnit[8] has been integrated into the standard tools to ease running of JUnit tests.

Java classes can be unloaded and then a possibly new version can be loaded again at run-time – a feature missing in other Java virtual machines. A new can be even written and then “accepted” in the class browser. A standard `javac` is then invoked a new class is reloaded into a running system. However, this feature is still experimental.

9. Related Work

9.1 JavaConnect and JNIPort

JavaConnect (Brichau and De Roover [1]) and **JNIPort** (Geidel [4]) are Smalltalk libraries that allow interaction with Java code from within Smalltalk. A Java virtual machine is linked into the Smalltalk virtual machine and the communication is made via foreign-function interfaces. Cross-language messages are translated into FFI invocations of either environment.

Java classes can be browsed, but cannot be modified from within Smalltalk. This is caused by the implementation of the Java virtual machine. Each Java object passed through FFI must be wrapped and registered, which generates additional overhead and disables automatic garbage collection of such objects. Because Java code is running in a separate virtual machine, proxy Java class must be generated for every Smalltalk class passed into Java. To reduce the overhead due to FFI calls, JavaConnect introduces the concept of language shifting objects. Shifted Java objects have part (or whole) of their behavior translated into Smalltalk, but not all instructions and language constructs are supported (such as `MONITORENTER` instruction and `synchronized` methods). In multithreaded applications, object state must be synchronized and problems arise, when a single native-threaded Smalltalk virtual machine interacts with a multithreaded Java application. Deadlocks can occur when a Java thread tries to communicate with the Smalltalk virtual machine, whose single native thread is blocked by another Java thread.

In STX:LIBJAVA, Java classes can be created, modified, or destroyed in runtime. There is no need to synchronize object state across two virtual machines. Java methods are directly executed, therefore no translation or interprocess communication is needed.

9.2 IBM VisualAge

IBM VisualAge for Java (Deupree and Weitzel [2]) includes an interaction mechanism on the Smalltalk side. Communication is realized using remote-method invocation (RMI). Java and Smalltalk virtual machines run in parallel, possibly even on two separate machines. The transition between languages is explicit and managed by the programmer. A lot of boilerplate code must be written and objects must be registered, converted and maintained by the programmer when crossing language barrier.

STX:LIBJAVA does not require any boilerplate code to access code across the language barrier, nor does it require any explicit handling or conversion when crossing the barrier. There is no need to set up an RMI service and to explicitly register objects in the RMI registry.

9.3 Redline Smalltalk

Redline Smalltalk (Ladd [9]) is a Smalltalk implementation running on the JVM. Java classes are dynamically compiled from Smalltalk source code and loaded into JVM using standard class loaders. Smalltalk exceptions are mapped onto Java exceptions. Therefore, it is not possible to proceed or retry an exception. This greatly restricts the number of Smalltalk applications and libraries which could run on Redline Smalltalk. Some methods, especially those related to object space reflection, such as `allInstances` or `become:`, are not supported for performance reasons. Common types such as string and integer must be explicitly converted when crossing the language barrier. Due to Java’s static type system, Smalltalk objects can only be passed as argument if the Java method expects that type.

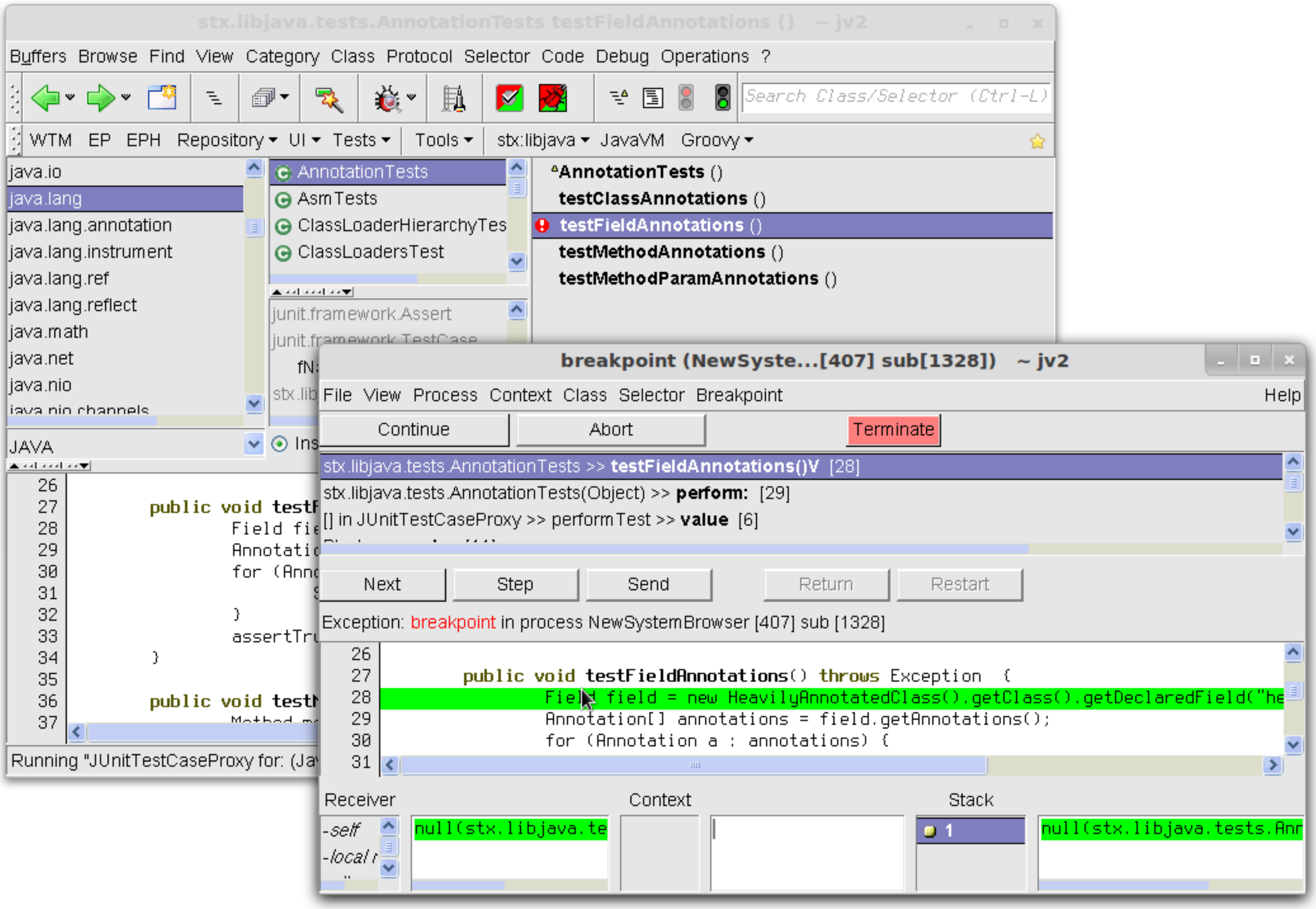
STX:LIBJAVA does not alter any feature of the Java language, Java reflection is also fully compliant with the original JVM. Any Java application could run on STX:LIBJAVA without modifications. Common types are automatically converted. Java and Smalltalk methods can be passed in as needed and the programmer is freed from the need to handle Java or Smalltalk objects differently. STX:LIBJAVA is currently the only language implementation of this kind on Smalltalk/X, however the approach does allow for direct communication between other such languages if they become relevant[9].

10. Conclusion And Future Work

In this paper, we have presented STX:LIBJAVA, a Java virtual machine implementation integrated into the Smalltalk/X environment. We have described the main problems of a seamless integration of Smalltalk and Java languages and their solutions. We use dynamic method proxies to allow for Java code to be easily called from Smalltalk and the other way round. Also, we described how to integrate Smalltalk and Java exception and synchronization mechanisms, so Smalltalk code can handle Java exceptions while the semantics of Java finally and synchronized blocks are preserved. A number of significantly large programs such as SAXON XSLT processor and Apache Tomcat Server/JSP container run on STX:LIBJAVA.

[8] www.junit,org

[9] Ruby (Vraný [11], Chapter 8) and a JavaScript dialect (Gittinger [5]) have been integrated into are Smalltalk/X, but these translate the source language into Smalltalk/X bytecode and do not suffer from the complexity resulting from major semantic differences of eg. the exception mechanism



```
stx.libjava.tests.AnnotationTests >> testFieldAnnotations()V [28]
stx.libjava.tests.AnnotationTests(Object) >> perform: [29]
[] in JUnitTestCaseProxy >> performTest >> value [6]
```

Exception: breakpoint in process NewSystemBrowser [407] sub [1328]

```
    public void testFieldAnnotations() throws Exception {
        Field field = new HeavilyAnnotatedClass().getClass().getDeclaredField("he
        Annotation[] annotations = field.getAnnotations();
        for (Annotation a : annotations) {
```

```
null(stx.libjava.te
null(stx.libjava.tests.Anr
```

Figure 3. Class Browser and Debugger showing Java code

In the future we plan to further improve the integration of Java into Smalltalk environment, for instance: add support for extension methods on Java classes, integrate Java support to Smalltalk/X packaging tools and building process, improve code highlighting and navigation, add more specialised object inspectors. We also plan to extend STX:LIBJAVA to provide a fully incremental development environment for Java, similar to that provided for the Smalltalk language.

Acknowledgement. We would like to gratefully thank to Oscar Nierstrasz and Eliot Miranda for their precious comments.

References

[1] J. Brichau and C. De Roover. Language-shifting objects from java to smalltalk: an exploration using javaconnect. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pages 120–125, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5. doi: 10.1145/1735935.1735956. URL http://doi.acm.org/10.1145/1735935.1735956.

[2] J. Deupree and M. Weitzel. Visualage integration for the 21st century: Smalltalk, java, websphere. URL http://www-01.ibm.com/support/docview.wss?uid=swg27000174.

[3] Expecco. Java interface library 2.1. http://wiki.expecco.de/wiki/Java_Interface_Library_2.1.

[4] J. Geidel. Jniport, Feb. 2011. URL http://jniport.wikispaces.com/.

[5] C. Gittinger. Javascript compiler and interpreter. http://live.exept.de/doc/online/english/programming/goody_javaScript.html.

[6] C. Gittinger. Die Unified Smalltalk/Java Virtual Machine in Smalltalk/X. *In Proceedings of NetObjectDays*, 1997.

[7] C. Gittinger and S. Vogel. Smalltalk/Java Integration in Smalltalk/X. *Tagungsband STJA'97, GI Fachtagung Objektoriente Softwareentwicklung*, 1997.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification, The (3rd Edition)*. Addison Wesley, Santa Clara, California 95054, U.S.A, 3 edition, 6 2005. ISBN 9780321246783. URL http://java.sun.com/docs/books/jls/.

[9] J. Ladd. Smalltalk implementation for the jvm. URL www.redline.st.

[10] T. Lindholm and F. Yellin. *Java^{TM} Virtual Machine Specification, The (2nd Edition)*. Prentice Hall, Santa Clara, California 95054 U.S.A, 2 edition, 4 1999. ISBN 9780201432947. URL http://java.sun.com/docs/books/jvms/.

[11] J. Vraný. *Supporting Multiple Languages in Virtual Machines*. PhD thesis, Faculty of Information Technologies, Czech Technical University in Prague, Sept. 2010.

[12] J. Vraný, J. Kurš, and C. Gittinger. Efficient Method Lookup Customization for Smalltalk. *Objects, Models, Components, Patterns*, pages 1–16, 2012. doi: 10.1007/978-3-642-30561-0_10. URL http://www.springerlink.com/index/PU875371770R562R.pdf.