SAND2011-6360C

$\mathcal{M}$*onarch*: A High-Assurance Java-to-java (J2j) Source-code Migrator[*]

Victor L. Winter, Jonathan Guerrero, Carl Reinke
University of Nebraska at Omaha
Department of Computer Science
{*vwinter, creinke, jguerrero*}*@mail.unomaha.edu*

James T. Perry
Sandia National Laboratories
Surety Electronics & Software
Department 2144
jtperr@sandia.gov

Abstract

JVM-based processors used in embedded systems are often scaled back versions of the standard JVM which do not support the full set of Java bytecodes and native methods assumed by a JVM. As a result, code bases such as Java libraries must be migrated *in order make them suitable for execution on the embedded JVM-based processor. This paper describes* $\mathcal{M}$onarch*, a high-assurance Java-to-java (J2j) source code migrator that we are developing to assist such code migrations.*

1 Introduction

At Sandia National Laboratories, a hardware implementation of the JVM [1] is being designed for use in resource-constrained embedded applications. This implementation has capabilities similar to the Java Card. Sandia Engineers have determined that the creation of applications for their platform would be significantly facilitated if a suitable subset of the libraries in the Java Standard Edition (SE) API could be made available to their embedded systems developers. This has given rise to a funded project whose goal is to develop the capability of migrating Java code (e.g., select Java libraries) to Java-based platforms in a highly reliable manner[1].

1.1 Contribution

This paper describes a novel approach and infrastructure for *Java-to-java (J2j) source code migration*, an area in the field of migration in which little to no work has been done. Related work has typically centered around J2X or X2J migrations where X is some language other than Java (e.g., C/C++ or .NET) [2][3][4]. Assumptions upon which our J2j migration is predicated imply that *deletion of code* plays a central role in J2j migration. Deletion requires analysis that is fundamentally different from translation-based analysis. In particular, we are not aware of any tool capable of performing the kind of dependency analysis necessary to safely justify deletion.

Our J2j migrator integrates transformation-based programming and function-based programming within a framework called the *TL System*. Complimenting this is Bascinet, a GUI developed in Java, providing system-level support for both developing our migrator and for then migrating Java libraries (e.g., Java code bases residing within folder hierarchies). Comprehension of source code is supported by several tools and artifacts including a special-purpose plugin written for Cytoscape. The resulting J2j source-to-source migration tool is called *Monarch*.

The remainder of the paper is as follows: Section 2 summarizes the restrictions of the family of processors we are targeting. Section 3 describes J2j migration and its goals. Section 4 gives an overview of $\mathcal{M}$*onarch*. Section 5 describes the infrastructure upon which $\mathcal{M}$*onarch* is built. And Section 6 concludes.

2 Software Development for Restricted JVM-based Platforms

From the perspective of *process*, developing code for the processor we are targeting is essentially identical to developing code for the JVM. Programmers can develop and debug programs on a desktop using an IDE such as Eclipse or Netbeans. Tools such as unit testers can be used to validate aspects of the software.

From the perspective of the *Java language*, software developed for a restricted processor may only contain features

[*]This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

[1]It should be noted, that the computational restrictions being considered are fairly representative of resource-constrained JVM's in general. Thus, the Java-to-java migration capability being developed can be used to target a variety of resource-constrained JVM's.

Java Language Restrictions

Feature	**Relevant Keywords**	**Status**
floating point	strictfp, float, double	unsupported unsupported
threading	synchronized volatile	unsupported unsupported
serialization	transient	unsupported
assertions	assert	unsupported
multi-dimensional arrays		unsupported

VM Restrictions

Feature	**Relevant Keywords**	**Status**
native methods	native	limited support
garbage collection		limited support
reflection		unsupported
(dynamic) class loading		unsupported

Table 1. A typical list of Java features not supported by restricted platforms.

supported by the processor. A typical list of features that are not supported by restricted processors is shown in Table 1. It should be noted that the use of native methods within a restricted processor is also limited. These feature restrictions and native method limitations extend to the entire code base of an embedded application, including any elements imported from standard Java libraries (e.g., `java.lang`). For example, the Reflection API is generally not available to the developer in part due to its native method dependencies. A somewhat different restriction applies to the method `finalize()`, used in the context of garbage collection, which is available to the Java programmer on a standard JVM. In particular, standard garbage collection is oftentimes not available within an embedded system.

3 An Overview of the J2j Migration Problem

Informally stated, the goal of J2j migration is to transform, at the source-code level, a code base such as a class library into a semantically equivalent form that has the additional property that it is also executable on a targeted platform. Ideally, the migrated and un-migrated versions of a code base would be indistinguishable to the user of the code base. Unfortunately, indistinguishability is a "tall order" and is generally not achievable in practice due to a variety of constraints (e.g., time and space) placed on embedded processors. As a result, concessions must be made. In particular, users must decide in which cases to accept reduced functionality (e.g., fewer methods) and when to accept altered functionality. Furthermore, these decisions must be made in the context of a larger system design where the properties of a migrated code base must be transparent to the design and development team.

There are two types of mechanisms that must be considered in the context of J2j migration: *removal* and *re-implementation*. Removal(-based migration) entails strict deletion of code that is not supported by the targeted platform. In contrast, re-implementation(-based migration) adapts code by replacing unsupported code fragments with equivalent (or near-equivalent) code fragments expressed in terms of computations supported by the targeted platform. It should be noted that in a practical setting, a straightforward re-implementation is not always possible.

3.1 Goals

Our J2j migration project has the following goals:

- **Correctness.** To produce migrated code that is correct with respect to the original code.
- **IV&V.** The code should be migrated in such a manner that migration can easily be subjected to independent verification and validation (IV&V).
- **Human Involvement.** Migration should be automated to the extent possible. However, the overall migration process should allow for human involvement such as the manual re-implementation of critical portions of a code base.
- **Repeatability.** The complete migration, including the substitution of manually developed code, should be replayable in a fully automatic manner.
- **Reuse.** Migration of a code-base (such as a Java library) may need to be repeated as new versions of the code-base are released. In this case, there should be support for reusing the re-implementations developed for the previous migration.

4 The $\mathcal{M}$*onarch* Migrator

Previously, we had developed a lightweight code migration tool implemented within a transformation system called HATS [5]. We considered our tool lightweight because dependency analysis it performed was not semantics-based. Instead, a simple syntactic matching algorithm was used to approximate dependency analysis. Interestingly enough, this approach yielded fairly good results [6]. Using this migrator we were able to automate a significant portion of what could be automated during the migration process. For example, our lightweight migrator achieved approximately a 91% coverage when applied to the `java.util` library. From an economic standpoint this could be considered a success.

This new migration tool is called *Monarch*[2]. Two major differences between the current and previous approach is that in the current approach we (1) no longer approximate dependency analysis, and (2) our transformation tool has been completely redesigned - in particular, a variety of higher-level as well as bookkeeping functions are now available though a single button click.

As shown in Figure 3, *Monarch* migration consists of two primary phases: *replacement* and *removal*. The replacement phase is manual and focuses on the re-implementation of code that in its original form has a dependency on an unsupported feature. In contrast, the removal phase is fully automatic and removes Java *elements* having dependencies on unsupported features. We define a Java *element* as a (1) field, (2) method, (3) constructor, or (4) initialization block.

4.1 Tool Capabilities

In addition to performing dependency-based removal, *Monarch* provides a framework in which it is relatively straightforward to gather a variety of metrics over large code bases (e.g., 250K LOC). Examples of metrics currently gathered include:

- The total source lines of code.
- The total number of classes, fields, methods, and constructors.
- The total number of initialization blocks and their location within the source code.
- The total number of occurrences of the `new` keyword within a constructor.
- The total number of single-type imports.
- The total number of static single-type imports.
- The total number of anonymous classes.
- The total number of class declarations occurring within a method or constructor. For example, the subset of the Java Standard Edition (SE) base libraries comprised of { `java.io`, `java.lang`,`java.math`, `java.nio`, `java.util` } contains 257,163 lines of code, has no class declarations within a constructor, and contains only one class declaration within a method.

[2]Butterflies are the archetype of transformation. Monarch butterflies are known for their migratory prowess, traveling roughly 2500 miles during their migration.

Metrics such as these have yielded empirical evidence that certain kinds of dependency "corner cases" are extremely rare.

Monarch also employs graphical representations of software. In particular, we are employing Cytoscape to help visualize element dependencies and subtype relationships within the code. We have developed a Java plugin for Cytoscape supporting a number of views on a targeted code base, including:

- **Standard View:** This view shows all the types in the code base, their members, and various dependencies among these entities. Color coding is used to visually distinguish both *unsupported entities* (e.g., the primitive type `double`) and *external entities* (i.e., entities lying outside of the (targeted) code base). Color coding is also used to identify *unsupported dependencies* (dependencies on unsupported entities) as well as *external dependencies* (dependencies on external entities).
- **Inheritance View:** This view shows the inheritance structure (i.e., both `extends` and `implements`) of all types in the code base. Shapes are used to distinguish class, interface, and enumerated types.
- **Structure View:** This view shows class and package membership. Data dependencies are not shown, but color coding is used to identify members having dependencies on unsupported entities.
- **Package Membership View:** This view shows class-in-package membership dependencies as well as class-imports-from-package dependencies.

Figures 1 - 2 shows each of the four views described for a targeted code base centering on `java.math`.

5 Infrastructure

Monarch is implemented within the TL System and Bascinet.

5.1 The TL System

TL is a special-purpose language we have developed for expressing transformation-based computation. TL is tightly integrated with the functional language SML. This provides a context for expressing computation in a hybrid fashion spanning transformation-based programming and function-based programming. It is in this programming landscape that *Monarch* is being developed.

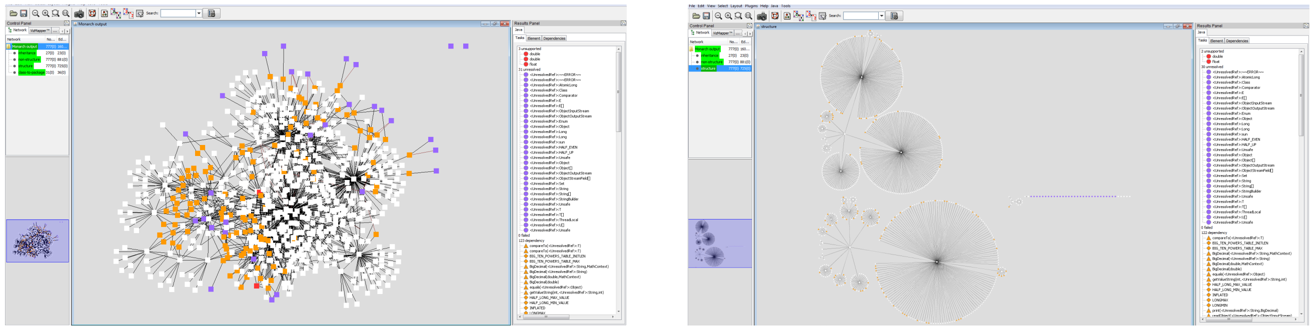

Figure 1. Standard and Structural Views

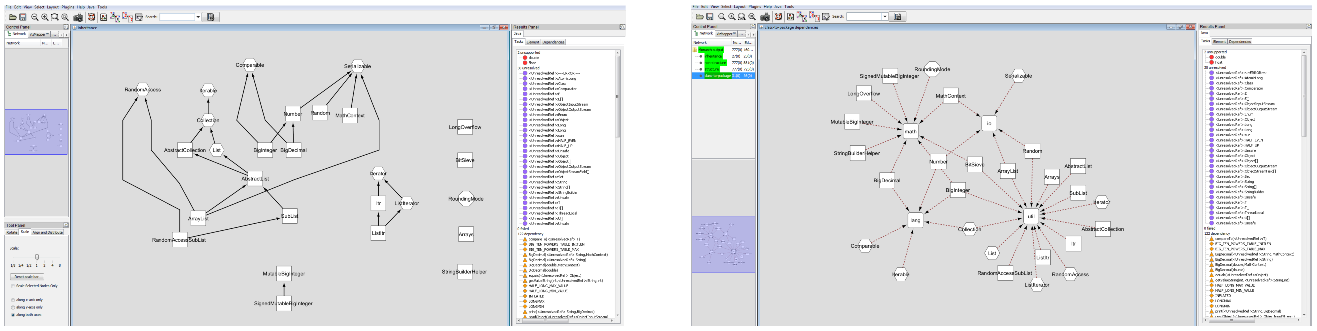

Figure 2. Inheritance and Package Membership Views

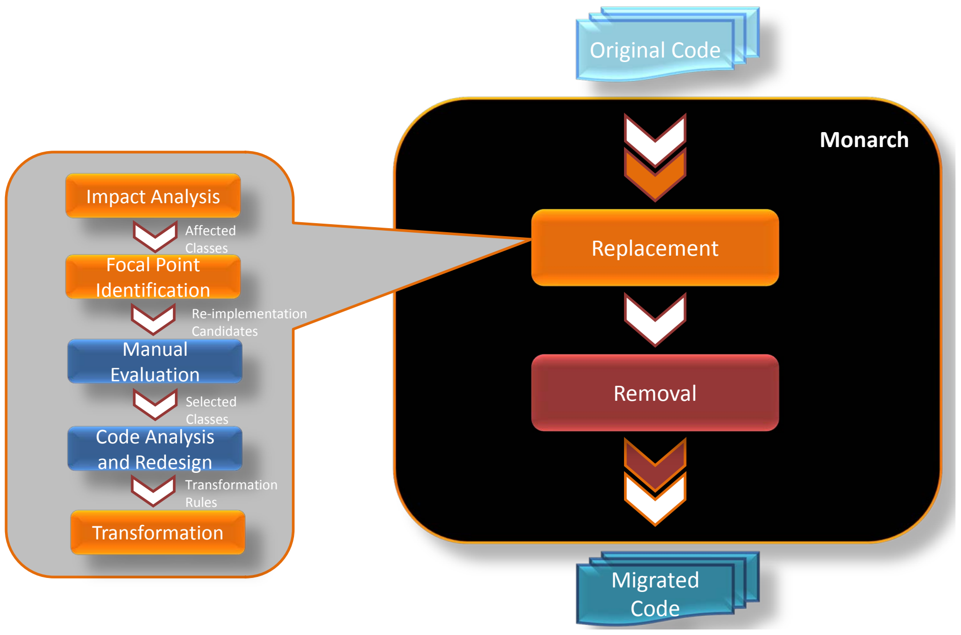

Figure 3. *Monarch*

The TL System includes a (1) GLR parser for translating plain text (e.g., ascii representations of programs) into terms, (2) a TL interpreter implemented in SML for rewriting terms, and (3) a powerful pretty-printer that can be used to translate terms into a variety of representations such as plain text documents and HTML documents.

The terms that TL transforms are parse trees. These terms contain hidden information describing their point of origin. This information includes the name of the file and the row and column number in the file. *Point-of-origin* information can be extremely useful for tracing information within a transformation. In the context of code migration, point-of-origin information can be queried to determine the source code location of fields, methods, and constructors having unsupported dependencies. Point-of-origin information can also be used to calculate the number of lines in a (plain text) file.

The TL System can be executed from the command line and runs on both the Windows and Unix operating systems.

5.2 Bascinet

Bascinet is a GUI, inspired by the HATS GUI (its predecessor), that is written in Java and provides support for the development and execution of TL applications. Conceptually speaking, "Bascinet is to TL" as "Eclipse is to Java".

Bascinet and the TL System are integrated in a manner that seamlessly supports the application of transformations (i.e., TL programs) to file hierarchies. A developer simply selects the transformation they want to apply together with the file or file hierarchy to which the selected transformation should be applied.

Bascinet supports two distinct application modes: (1) a *discrete mode application* in which the selected transformation is applied in a repetitive manner to each file in a file hierarchy, and (2) a *continuous mode application* in which the selected transformation is applied a single time to the entire contents of a file hierarchy.

Continuous mode application is very useful when the entity to be transformed has been distributed across a number of files. For example, using this application mode it is straightforward to develop transformations for gathering a wide variety of metrics over a Java code base.

Bascinet allows the developer to control to which file extensions (e.g., dot-java) a transformation should be applied. From a practical standpoint, this is an important feature when applying a transformation to a folder hierarchy consisting of hundreds of folders and thousands of files. For example, Java libraries occasionally contain files having extensions other than the dot-java extension. Applying a Java-oriented transformation to a non-Java file will result in failure. The ability to exercise extension-based control over transformation application permits transformations to be applied directly to Eclipse workspaces. For example, migrator test suites can be developed in Eclipse and then validated in $\mathcal{M}$*onarch* without any modification to the folders generated by Eclipse.

6 Summary and Conclusion

When developing applications for restricted JVMs it is beneficial to leverage the functionality provided by standard Java libraries. In order to make this possible, a Java-to-java (J2j) migration is required. Resource constraints on embedded processors can prohibit exotic migrations such as datatype emulation. Thus, J2j migration includes removal as well as re-implementation of code. The analysis needed to determine what must be removed is automatable and is highly complex.

$\mathcal{M}$*onarch* is a J2j migrator being developed within the TL system in which: (1) re-implementation is assisted by visual code representations such as the views we have developed within Cytoscape, and (2) the removal phase of migration is fully automated and is based on a complete dependency analysis. Re-implementations are encapsulated as transformations and the resulting migrations can then be replayed and readily subjected to IV&V.

References

[1] T. Lindholm and F. Yellin, editors. *The Java Virtual Machine (Second Edition)*. Addison-Wesley, 1999.

[2] J. Martin. *Ephedra - A C to Java Migration Environment: Approaches, case studies and tools for migrating legacy systems from C and C++ to Java*. Lambert Academic Publishing, 2011.

[3] J. Martin and H. A. Müller. Strategies for Migration from C to Java. In *Proceedings of the 5th European Conference for Software Maintenance and Reengineering*, Lisbon, Portugal, 2001.

[4] I. Tilevich. Translating C++ to Java. *First German Java Developers' Conference Journal*, 1997.

[5] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.

[6] V. L. Winter, A. Mametjanov, S. E. Morrison, J. A. McCoy, and G. L. Wickstrom. Transformation-based Library Adaptation for Embedded Systems. In *Proceedings of the* 10^{th} *IEEE International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2007.