SAND2011-3980C

Monarch: 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 Monarch, 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 [4] 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 Process Automation

Software migration based on a purely manual process is typically an extremely labor intensive endeavor. As a result, it is often cost effective to develop tools that (at least partially) automate the migration process. In this context, it is also worth noting that software migration is a process which typically cannot be fully automated [6]. In practice, this concession provides the justification for the development of tools which either automate only a portion of what can be automated or which provide approximations to what can be automated.

On the subject of automation it has been argued that a tool capable of automatically achieving a 75% migration rate produces significant cost savings over a purely manual approach [1]. However, it will also then generally be the case that the core complexity of the migration process will be found in the software *pool* encompassing the remaining 25%. There are two primary reasons for this pooling phenomenon: *economic* and *theoretical*. Economic forces provide an incentive for expanding tool capabilities until a point of diminishing return is reached; at which time tool development may slow dramatically or even stop. In contrast, theoretical forces provide hard limits to what can be achieved.

When high-assurance (and not economics) is the primary driver behind the implementation of tools supporting process automation, then automation should leveraged to the fullest extent theoretically possible. Instead of settling for achieving 75% of *what could be automated* one must strive for achieving 100% of *what can be automated*. It should also be noted that some goals within a process may not be fully automatable due to theoretical limitations. Therefore, when we say "100% automation", we are talking about fully automating that portion of a process that can be automated.

1.2 The 99/1 Pooling Phenomena

As automation approaches 100%, tool capabilities are expanded to handle operational profiles of the problem space that rarely occur. Generally speaking, these spaces (what we are metaphorically calling *pools*) are poorly understood. In practice, it is here where a considerable por-

*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.

tion of the challenges to high-assurance tool development are encountered. For example, our effort in developing a J2j source-code migration tool has led to exploration of corner cases of what is technically possible within the Java language. This exploration has led to the discovery of legal Java code revealing bugs [10] in the static analysis and refactoring functions of Eclipse, Netbeans, and IntelliJ.

In a variety of fields of study the *pooling phenomenon* associated with a particular attribute, such as complexity, is referred to in broad terms as the Pareto Principle[2] otherwise known as the 80/20 rule. However, our experiences with high-assurance tool development suggest that 80/20 is a misleading characterization of this phenomenon. We think that for complex high-tech systems a better quantification would be something like 99/1. Evidence consistent with this quantification has also been presented by others. On October 2, 2002 Microsoft CEO Steve Ballmer wrote a 3-page memo which contained the following:

> *"One really exciting thing we learned is how, among all the software bugs involved in reports, a relatively small proportion causes most of the errors. About 20 percent of the bugs cause 80 percent of all errors, and-this is stunning to me-one percent of bugs cause half of all errors."[2]*

The significance and challenges that the *99/1 pooling phenomenon* poses to the construction of high-assurance tools should not be underestimated. Initial prototypes may implement algorithms that correctly handle large portions of the problem space, but which can break in fundamental ways when an attempt is made to extend these algorithms to the entire problem space. This poses a hard-to-quantify risk to software engineering approaches based on incremental development. Specifically, incremental development can exhibit a non-linear behavior: in the sense that the level of effort required to achieve the next increment is not linearly related to the size of the increment. In other words, correctly implementing the next small increment can require an unexpectedly large amount of effort - perhaps even triggering a fundamental re-design of the tool. A lesser such tipping point is encountered at the 75-80 percentile range as was mentioned a the beginning of this section.

1.3 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). 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 overviews related work. Section 3 summarizes the restrictions of the family of processors we are targeting. Section 4 describes J2j migration and its goals. Section 5 gives an overview of how $\mathcal{M}onarch$ approaches J2j migration as well as some of the challenges encountered. Section 6 describes the infrastructure upon which $\mathcal{M}onarch$ is built. And Section 7 concludes.

2 Related Work

It has been recognized that language conversion/migration is a hard and risky endeavor [11]. In [11], the authors mention they have first hand knowledge of companies that have gone bankrupt as a result of underestimating the difficulty of language migration.

In the literature, a variety of code migration projects have been reported. These migration efforts have spanned a wide spectrum of languages including: Smalltalk to C [15], C/C++ to Java [5][6][12], Java to C#/.NET [9], PL/I to C++ [3], and Cobol to Visual Basic [11]. The motivation behind such migrations center around issues such as (1) reengineering and evolution of legacy systems, (2) efficiency, (3) integration with other technologies, and (4) corporate-level decisions to shift technologies. The migration tools that have been developed typically have caveats limiting either their generality, safety/reliability, or level of automated support they provide.

Of the migrations listed in the previous paragraph, the C/C++ to Java migrations are the most closely related to our migration efforts. A fairly comprehensive discussion of approaches to C/C++ to Java migration can be found in [6]. In essence, C/C++ code can be executed on a JVM in one of three ways: (1) as native methods that extend the JVM, (2) as compiled byte codes which are then executed by the JVM, or (3) as translated (C to Java) source code which

[2] In 1906 the Italian economist Vilfredo Pareto observed that 80% of the land in Italy was owned by 20% of the population[8].

can then be compiled using a standard Java compiler. The first approach utilizes the Java Native Interface (JNI) to provide the mechanism for transferring data to-and-from native methods and methods residing within class files. In this approach, C-based computations are executed outside of the JVM. This circumvention of the JVM introduces a security risk which may be unacceptable in high-consequence applications. In the second approach C code is compiled into byte code which can then be executed on the JVM. The *Java Backend for GCC* is a migrator that takes this approach. Conceptually, the migration uses a single Java array to hold the variables in the C program. A negative consequence of this is that this structure circumvents Java's type safety which again may introduce an unacceptable risk for high-consequence applications.

A migration strategy based on source-code level transformation of C/C++ to Java is taken by a number of tools including: Ephedra [5], C2J++ [12], and Novosoft C2J. Of these, Ephedra appears to be the most comprehensive, though migration is also not fully automated. In general, issues that make C/C++ to Java migration hard include: (1) pointers, (2) memory management, (3) call-by-reference vs call-by-value parameter passing, (4) datatype conversions, and (5) goto statements.

Regardless of which level of abstraction one selects for migration, the resulting code and/or associated interfaces make code review by an IV&V team difficult. In the high-consequence arena this would then require other mechanisms to be used to provide high assurance in the proper functioning of migration tools as well as the results of individual migrations.

3 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 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

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.

`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.

4 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. The description in Figure 1 provides a more formal statement of the J2j migration problem.

There are two types of mechanisms that must be considered in the context of J2j migration: *removal* and *re-*

1. Given:
   (a) a set of elements constituting a code base:
   $$\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$$
   (b) a set of unsupported elements
   $$\mathcal{U} = \{u_1, u_2, \ldots, u_k\}$$
2. Develop a code base $\mathcal{C}''$ such that:
   (a) $\mathcal{C}''$ maximally preserves the code base of $\mathcal{C}$
   (b) $\mathcal{C}''$ has no dependencies on $\mathcal{U}$

Figure 1. Statement of the J2j migration problem

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.

From the perspective of computability, removal and re-implementation are distinct. Removal is algorithmic and as a result lends itself to automation. On the other hand, re-implementation in its full generality is non-algorithmic and thus cannot be fully automated. Re-implementation requires a human-in-the-loop. Because of this, re-implementation has an error-prone dimension to it centering around the manual development of (new) code. Thus, re-implementation can entail a potentially unacceptable risk of introducing bugs into the migrated code base (and ultimately the embedded system design).

Aside 1 *With respect to the assurance argument it is important to note that the Java Base Libraries, which are the target of our migration efforts, typically undergo a significant number of tests before their official release by Sun. These libraries are also subjected to a maturation process based on bug feedback from a large group consisting of millions of users. In contrast, re-implemented components of libraries used in embedded applications (especially one-of-a-kind embedded applications) will not be subjected to the maturation process associated with a large user base. Thus, caution should be exercised when making decisions to re-implement portions of a library.*

From the perspective of assurance, library migration based on *removal* is more attractive than migration based on *re-implementation*. In theory, removal can be precisely defined as a *relation* between unsupported computational elements and code fragments. In practice, it is the calculation of this relation and its transitive closure that gives rise to the complexity of removal-based migration.

4.1 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 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.

5 The *Monarch* Migrator

Previously, we had developed a lightweight code migration tool implemented within a transformation system called HATS [13]. 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 [14]. 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.

Our current goal is to enhance this general approach to migration, producing a tool in which the automatable portion of migration is fully realized (i.e., automated at 100%). This new migration tool is called *Monarch*[3]. Two major differences between the current and previous approach is that in the current approach we (1) no longer approximate

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

dependency analysis, and (2) our transformation tool has been completely redesigned.

As shown in Figure 4, *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.

5.1 The Replacement Phase

The replacement phase begins with an inspection of the code in order determine the dependencies of its elements. A variety of tools and artifacts are being developed in order to facilitate comprehension of the dependencies within the targeted code base. Numerous metrics can be collected and displayed in symbolic form. For a targeted code base, examples of metrics include, but are not limited to:

- 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 Standard Basis library consists of 257,163 lines of code, contains no class declarations within a constructor, and contains only one class declaration within a method.

Monarch also employs graphical representations of data. In particular, we are employing GraphViz and 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 show inheritance as well as class membership. Data dependencies are not shown, but color coding is used to identify members having dependencies on unsupported entities.
- **Unsupported/External View:** This view shows only the portion of the code base having unsupported or external dependencies.

Figures 2 - 3 shows each of the four views described for a targeted code base centering on `java.lang`.

After an initial code inspection, an exploratory removal-based migration can be performed. This migration is fully automatic and typically ends up removing too much of the targeted code base.

Example. Every constructor of the class `Throwable` has a dependency on a native method called `fillInStackTrace`. If this native method is not supported on the platform being targeted, then unsupported dependencies will extend to all `Exception` and `Error` classes. In turn, all methods or constructors that throw exceptions will also have unsupported dependencies. As a result, removal-based migration will yield a code base that is near empty.

The goal of the replacement phase is to manually re-implement code in order to remove unwanted dependencies from "must have" portions of the target code base. These re-implementations are then recorded as transformations $\xrightarrow{replace}$ which can then be (automatically) applied prior to removal-based migration.

$$\mathcal{C} \xrightarrow{replace} \mathcal{C}'$$

5.2 The Removal Phase: $\mathcal{C}' \xrightarrow{remove} \mathcal{C}''$

In *Monarch*, the removal phase is fully automated. In this phase, fields, methods, and constructors having external or unsupported dependencies are removed. As was mentioned in Section 1.3, removal comes with its own set of challenges. This section discusses some of the challenges to

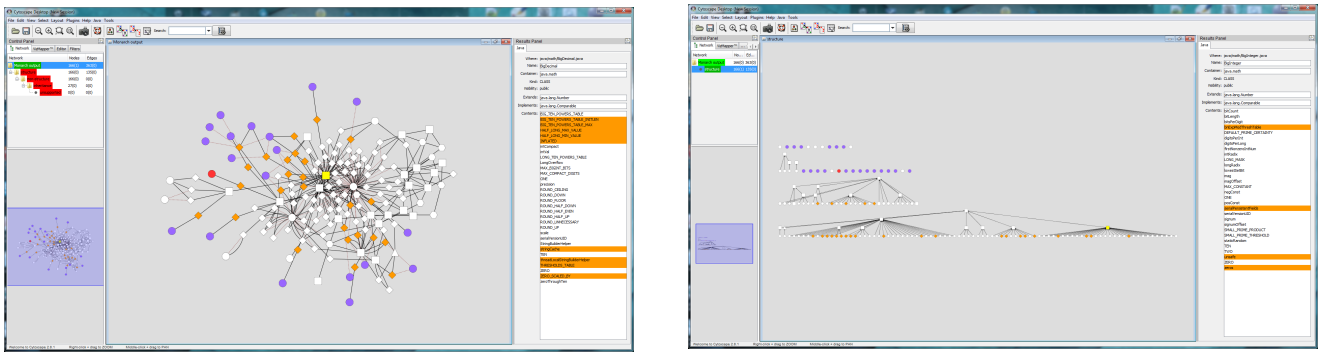

Figure 2. Standard and Structural Views

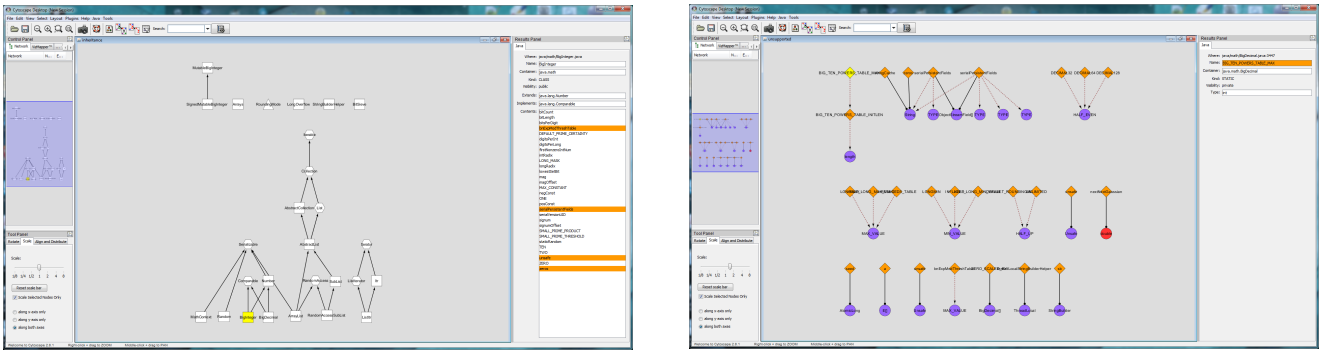

Figure 3. Inheritance and Unsupported/External Views

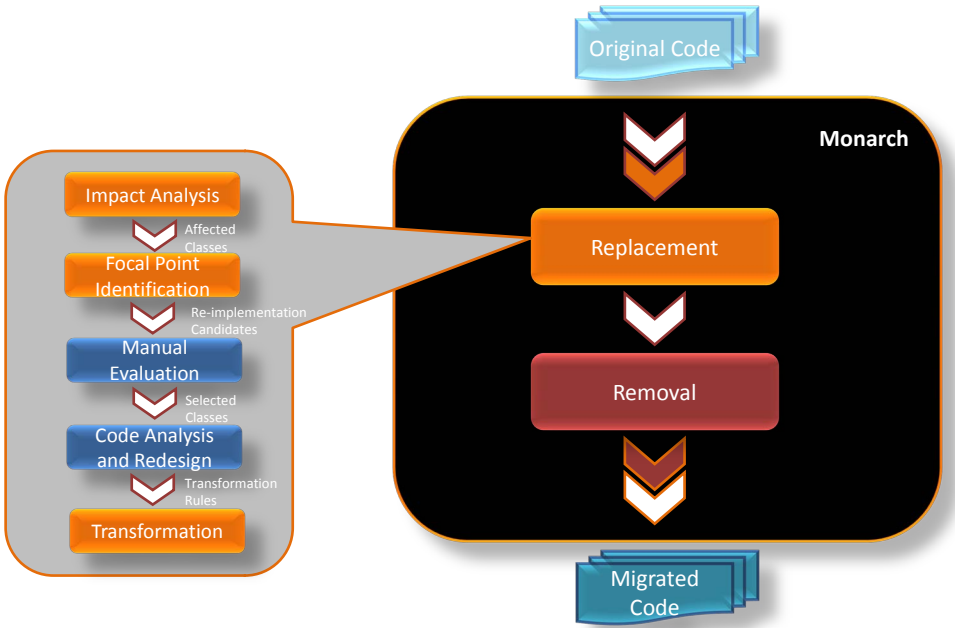

Figure 4. *Monarch*

performing dependency analysis in the context of removal. In particular, removal can have far reaching consequences.

5.2.1 Resolution Complexity

Resolution is a relation between references and declarations. Java's resolution algorithm is extremely complex and encompasses a substantial number of special cases. It is beyond the scope of this article to give a detailed discussion on this topic. Nevertheless, we would like to give one example highlighting our claim. Figure 5 shows a Java code fragment in which there are multiple declarations of the class `C2`. Within the inner class `B2`, a reference to `C2` is made. In this case, the canonical form of `C2` is `extra.C2`. That is, the *reference* `C2` *resolves* to `extra.C2`. It is worth noting that the private inner class `C1.D1` declared in `B1` is visible from within `B2`, but the private inner class `C2` declared in `B1` is not visible from within `B2`.

```
package p;
import extra.C2;

public class A {
    class B1 {
        class C1 {
            private class D1 {}
        }
        private class C2 {}
    }
    class B2 extends B1 {
        C1.D1 x0;
        C2 x1; // canonical form of C2 is extra.C2
    }
}
```

```
package extra

public class C2 {}
```

Figure 5. An example of a complex resolution.

5.2.2 Dependency Analysis Challenges Posed by Upcasting

Upcasting can obfuscate dependency analysis, which can be extremely dangerous within the context of removal. Consider the following expression involving the conditional creation of an object followed by a call to the method `foo` on the created object.

$$(x < y : newA1() : newA2()).foo()$$

In a code fragment such as this, it is possible for the method `foo` to have overriding declarations in both `A1` and `A2`. In this case, dependency analysis must capture the set of possible methods being referred to.

5.2.3 Dependency Analysis Challenges Posed by Overriding

Consider a class hierarchy in which every subclass in the hierarchy has an overriding declaration for a particular field x. Furthermore, suppose that lowest subclass Z in this hierarchy is an inner class whose enclosing class also has a declaration for the field x. Lastly, suppose that the field x is also statically imported into the compilation unit in which the lowest subclass Z resides. In this case, the removal of the field x from the body of the class Z will change the functionality of the code. To prevent this, all secondary resolution possibilities must be prohibited! A discussion of how this can be done lies beyond the scope of this paper.

6 Infrastructure

Monarch is implemented within the TL System and Bascinet.

6.1 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.

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.

6.2 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.

7 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] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49:275–291, 2007.

[2] S. Ballmer. http://www.microsoft.com/mscorp/execmail/2002/10-02customers.mspx.

[3] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Mueller, and J. Mylopoulos. Code migration through transformations: An experience report, 1998.

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

[5] 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.

[6] 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.

[7] J. Martin and H. A. Müller. C to Java Migration Experiences. In *Proceedings of the 6th European Conference for Software Maintenance and Reengineering*, Budapest, Hungary, 2002.

[8] V. Pareto. *Manuale di Economia Politica*. Piccola Biblioteca Scientifica, Milan, 1906.

[9] W. Partners. Case Studies on Platform Migration. Technical Report ITEA-04032, ITEA, 2008.

[10] J. T. Perry, V. Winter, H. Siy, S. Srinivasan, B. D. Farkas, and J. A. McCoy. The Difficulties of Type Resolution Algorithms. Technical Report SAND2010-8745, Sandia National Laboratories, December 2010.

[11] A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17:111–124, 2000.

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

[13] 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.

[14] 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.

[15] K. Yasumatsu and N. Doi. SPiCE: A System for Translating Smalltalk Programs Into a C Environment. *IEEE Transactions on Software Engineering*, 21(11):902–912, 1995.