SAND2015-0701J

Java Source Code Analysis for Library Migration to Embedded Systems

Victor Winter[1], James A. McCoy[2], Jonathan Guerrero[1], Carl Reinke[2], and James T. Perry[2]

[1]Department Of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182 USA
[2]Sandia National Laboratories, Albuquerque, NM

Abstract

Embedded systems form an integral part of our technological infrastructure and oftentimes play a complex and critical role within larger systems. From the perspective of reliability, security, and safety, strong arguments can be made favoring the use of Java over C in such systems. In part, this argument is based on the assumption that suitable subsets of Java's APIs and extension libraries are available to embedded software developers. In practice, a number of Java-based embedded processors do not support the full features of the JVM. For such processors, source code migration is a mechanism by which key abstractions offered by APIs and extension libraries can made available to embedded software developers.

The analysis required for Java source code-level library migration is based on the ability to correctly resolve element references to their corresponding element declarations. A key challenge in this setting is how to perform analysis for incomplete source-code bases (e.g., subsets of libraries) from which types and packages have been omitted. This article formalizes an approach that can be used to extend code bases targeted for migration in such a manner that the threats associated the analysis of incomplete code bases are eliminated.

Index Terms

source code analysis, Java, JVM, embedded system, Library migration

Manuscript received September 1, 2014. Corresponding author: V. Winter (email: vwinter@unomaha.edu).

CONTENTS

I. Introduction

Embedded systems form an integral [42] and often critical part of our technological infrastructure. Their use spans a growing number of application domains many of which operate in environments having significant requirements relating to reliability, security, and safety. Typical examples of domains having such requirements include: military, medical, energy, aerospace, and automotive.

In addition to their increased use, the complexity of embedded systems is also on the rise. As a result, there is a growing need to provide *ecosystems*, encompassing software, hardware and tool chains, capable of leveraging state-of-the-art practices in embedded system design and development. Ritzdorf has reported that for medical devices embedded platforms are very often "single-purpose, proprietary systems, built from the ground up with little to no dependence on common programming frameworks or operating systems" [38]. In a somewhat similar vein, a compilation of data [6] from subscribers to *Embedded Systems Design* during the time period 1997–2012 revealed the following:

- The favorite tools are compilers, oscilloscopes, and debuggers.
- The dominant language is C.
- Approximately 60-70% of the efforts is spent writing code.
- Java is used very little.

The fact that the compilation of the *Embedded System Design* subscriber data revealed such little use of Java is somewhat ironic since Java was originally intended for use in embedded systems [40]. From the perspective of reliability, security, and safety, strong arguments can be made favoring Java over C [37]. These arguments include: (1) platform independence, (2) type safety, (3) object orientation, (4) memory management, and (5) the abstractions offered by Java's APIs and extension libraries. On the other hand, the raw speed and low-level control (e.g., pointers) provided by C make C attractive for a variety of embedded systems – especially those with heavy real-time constraints.

For complex high-consequence embedded systems Java is (and should be) an attractive language. This is especially true in environments where changes in the underlying hardware are expected. Objections to the use of Java in embedded systems typically involve concerns over its memory footprint, execution speed, and the nondeterministic temporal properties of its garbage collection. However, significant inroads to addressing such concerns are being made as the result of advances in hardware as well as increased capabilities regarding how Java programs can be compiled for use in embedded systems [2][22][36].

Reddy [36] identifies Java's "very rich set of APIs and extension libraries" as one of the "great strengths" of Java. From the perspective of embedded systems, the problem with APIs and extension libraries is the memory footprint they entail. Reddy goes on to identify two approaches that can be used to mitigate this memory footprint problem: (1) *profiles* – which are subsets of Java APIs developed for specific application domains, and (2) *smart linking* – which strips from an API those classes, fields, and methods that are not used by the application.

A. Context of Our Research

$\mathcal{M}onarch$ [49] is a Java source code migration tool that is being developed at UNO to assist in migrating Java Core Libraries to the SCore platform, a hardware implementation of a subset of the JVM [24] being designed at Sandia National Laboratories. Figure 1 shows how applications, whose initial development takes place on a desktop, are compiled and translated into a file called a ROM image suitable for execution on the SCore.

The decision to develop the capability of migrating libraries rather than pursuing a *ground-up implementation* of the functionality found in the library has several benefits: (1) libraries that are in widespread use, such as the Java Core Libraries, have effectively been subjected to extensive "testing" and as a result are generally considered to be more reliable and mature than one-of-a-kind implementations that, by comparison, have been minimally tested, (2) an approach involving library migration can stay current with updates associated with the library, and (3) a source-code migration tool can be reused to migrate other libraries. That being said, it is important to note that library migration shifts the reliability burden

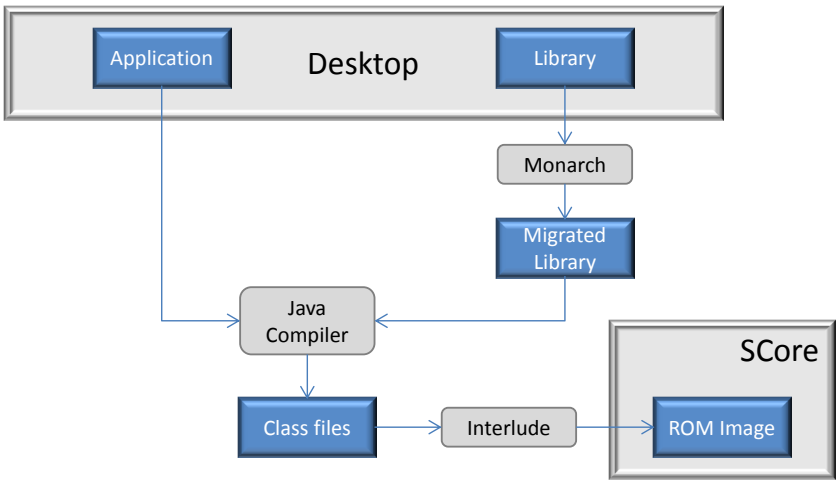

Figure 1. Software development for the SCore.

from the ground-up one-of-a-kind implementation of the library functionality *to* a general migration process and tool. As a result, providing high-assurance that the migration process and tool are correct is essential.

At this time, the primary code base targeted for migration to the SCore platform is a set of Core Libraries belonging to the Standard Edition (SE) of the Java Platform. A detailed description of the targeted code base is given in Appendix A. Further discussion on the subject of incorporating libraries to the SCore can be found in [52]

In its totality, $\mathcal{M}onarch$ migration is comprised of the following three stages.

1) **Re-implementation Stage** – In this stage, the target code base is manually inspected and functionality essential for a given SCore application is identified. If necessary, this essential functionality is re-implemented so that the resulting code is suitable for execution on the SCore. In practice, re-implementation is different from the ground-up customized library implementation in the following ways: (1) the functionality designated for re-implementation should be minimal comprising only a small percentage of the total migrated code base, and (2) re-implementation should only involve simple modifications to the existing code and thus present no significant assurance threats. The resulting re-implementations are then encoded as program transformations which can be automatically applied during the course of $\mathcal{M}onarch$ migration. In this article, we assume the target code base we are working with has passed through the re-implementation stage.
2) **Preparation Stage** – The purpose of this semi-automated stage, discussed in Section VI, is to obtain a *prepared code base* that is amenable to static analysis. For example, type declarations may be added to the code base for the purposes of completing subtype hierarchies.
3) **Removal Stage** – This is a fully-automated stage in which source-code unsuitable for execution on the SCore is stripped from the code base. The specification of such removals are in the form of *conditional rewrite rules*. A noteworthy feature of these rules is that their (application) conditions can make use of semantic properties of the Java source-code base[48].

B. Assumptions Upon Which This Research Rests

Fundamental to static analysis is the ability to determine the mapping between element *references* and element *declarations*. In this article, we use the term *resolution* when referring to this function. We *assume* that when given a complete code base, $\mathcal{M}onarch$ can correctly perform resolution [47]. Section II discusses how resolution information is obtained by a variety of static analysis systems and also gives a rational for $\mathcal{M}onarch$'s decision to implement a source code-level resolution function.

C. Contribution

The focus of this article is on identifying and addressing *additional challenges* to resolution arising from the consideration of *incomplete code bases* from which *top level types* as well as entire packages may have been removed. The Java Language Specification [18] defines a top level type as a top level class (Chapter 8) or a top level interface (Chapter 9). The descriptor *top level* implies that the type declaration is not nested within another type. Unless accompanied by addition removals, the removal of a top level type can cause resolution to produce incorrect results. For example, a reference that resolves to the type `p1.T` in the original (i.e., complete) code base could resolve to the type `p2.T` in the incomplete code base.

In practice, an incomplete code base, which we also call a *target code base*, is the source-code of a subset of a library that has been selected for migration to the SCore. Furthermore, the constraints surrounding code migration for the SCore are such that library migration *must* deal with such incomplete code bases (for more on this see Section V-E) so this issue cannot be side-stepped.

The key threat posed by incomplete code bases is that the *absence of declarations* can create conditions where the resolution of references within an incomplete code base will differ from corresponding resolutions within the complete code base. Our investigation identifies a number of closure properties that a code base must satisfy in order for $\mathcal{M}onarch$'s resolution function to yield correct results. *Supertype closure* is an example of one such property. For a code base to satisfy the *supertype closure* property it must be the case that *if* a type is in the code base, *then* its supertype must also be in the code base. It should be noted that in order to satisfy a closure property, the target code base must be extended (i.e., its size increased).

Due to practical limitations, closure properties cannot be used to eliminate all resolution problems associated with a target code base. The major problem here is that closures increase the target code base to which closures must again be applied. The only source-code base that is truly safe for analysis is one where closures have been exhaustively applied (i.e., until the fixed point is reached). In practice, such code bases are extremely large, and the SCore prohibits their consideration for migration. The Java compiler sidesteps this issue through a resolution function whose search capabilities are extensive and exotic. Its search for a resolvent can extend far beyond the code currently being compiled and includes the consideration of class files and jar files residing on the desktop and beyond. In contrast, $\mathcal{M}onarch$'s resolution function, for assurance purposes, is strictly limited to the source-code to which it is applied, and thus the problem of obtaining a code base suitable for analysis whose size is somehow proportional to the original target code base is an issue that must be confronted. In $\mathcal{M}onarch$ we address this issue through a novel concept called a *code skeleton* which extends the domain of the resolution function to source code *outside* of the target code base in a limited fashion. The key idea is that declarations in a skeleton can be searched by $\mathcal{M}onarch$'s resolution function, but the code within a skeleton will not be subjected to further analysis. Code skeletons, which are not migrated, thereby effectively limit the size of the code base that must be subjected to analysis. And finally, code skeletons can be automatically produced by $\mathcal{M}onarch$ through a set of program transformations and thus require little effort to create.

The combination of closure properties and code skeletons represent modifications to the target code base that must be performed in the preparation stage (i.e., stage 2) of migration in order to assure the correctness of resolution. We refer to a code base that has been modified in this fashion as a *prepared code base*. The goal of this article is to provide strong evidence that performing resolution on a prepared code base will yield correct resolution results for that code base.

The remainder of this article is as follows: Section III discusses related work. Section IV-B gives an overview of the SCore platform. Section V describes resolution. Section VI, describes how a target code base C, that is incomplete, can be *prepared* to assure the correctness of $\mathcal{M}onarch$'s resolution function, and Section VII concludes.

II. Class files versus Source Code

The implementation of a function capable of performing resolution by directly analyzing source-code is a major undertaking. As a result, an alternative taken by a number of Java static analysis tools, such as FindBugs [21], JLint [23] and Clousot [16], is to perform static analysis on Java class files. In this approach, information provided by the Java compiler forms the basis for certain kinds of analysis. An additional challenge for such frameworks is to map (or project) information mined from class files onto the Java source-code from which the class files were derived. It is at the source-code level that analysis results are typically communicated to the tool user.

Unfortunately, class files do not capture all the information present in source code. In [26] the pros and cons associated with the analysis of low-level (e.g., bytecode) versus program source code are discussed. Rutar et al. [39] investigate the effectiveness of a Bandera, ESC/Java, FindBugs, JLint, and PMD at finding various classes of bugs. In practice, information missing from class files can provide valuable insight into programmer intent which can improve both the false positive and false negative rates of static analysis. In recognition of the limitations of class file analysis, a growing number of static analysis tools, including Coverity SAVE [11][20][5], DMS [4][13][14], and ESC [17][9][10] are developing static analysis infrastructures that are predominantly centered on the analysis of source code. To be concise, even though the analysis performed by these tools is predominantly at the source-code level, these tools still interact with class files in order to obtain name and type information produced by the compiler's resolution function. Some theorem prover-based tools, such as Cibai [25], perform analysis exclusively [8] at the source-code level.

$\mathcal{M}onarch$ falls into the general category of static analysis (and manipulation) tools. However, $\mathcal{M}onarch$'s purpose is distinctly different from the tools mentioned in the previous paragraphs. $\mathcal{M}onarch$ migration centers around a dependency analysis-based *removal* of field, method and constructor declarations as well as analyzing the consequences of such removals. Furthermore, the scope and nature of the applications to which the SCore platform was being considered drove the need to develop general and completely unrestricted resolution analysis capabilities. Such analysis includes consideration of visibility issues relating to shadowing, overwriting, and overloading. A key question that $\mathcal{M}onarch$'s analysis must be able to answer can be stated as follows:

> For a given context, does the removal of a declaration expose (to this context) a secondary declaration that was previously not visible?

One key concern driving the $\mathcal{M}onarch$ analysis, described in this article, involves the set of public and protected declarations in code bases (e.g., subsets of libraries) targeted for migration. Specifically, it is references to this set that are available to embedded applications using these migrated libraries. A restriction placed on $\mathcal{M}onarch$ is that embedded applications, interacting with migrated library components, cannot themselves be subjected to analysis [50]. For this reason, $\mathcal{M}onarch$ must make conservative approximations with respect to removal. The analysis needed to answer such questions is predicated on the ability to perform resolution in an environment whose declarations change over time and thus cannot be realistically guided by (static) name and type information obtained from class files. Because of these issues and considerations (lesser issues exist as well), $\mathcal{M}onarch$ does not mine class files for resolution-based information. And for reasons resonating with what was mentioned in the previous paragraphs that $\mathcal{M}onarch$ analysis is performed on Java source code rather than on class files.

Assuming that a resolution function is correct is a fairly major assumption. We acknowledge that in practice most references can be resolved using a simple resolution algorithm [51]. However, in theory references can be created whose resolution is extremely complex. Such references tend to exercise the full capabilities of Java's resolution function and/or involve special cases. While it is true that such references are unlikely to occur in normal operating environments, such assumptions should not be made in high-consequence domains where abnormal or malicious behavior must be taken into account. In [33] a number of examples are given that reveal bugs in the resolution algorithms used by common Java editors and

```
1 package stackoverflow;
2 import static stackoverflow.A00.*;
3 public class A00 extends B00
4 {
5     public static class B00{}
6 }
```

Figure 2. A small program causing a stack overflow in the Java 1.6.0_26 compiler[47].

refactoring tools.

To mitigate concerns regarding the correctness of in-house developed resolution capabilities, we have created a test set which we use to automatically certify $\mathcal{M}onarch$'s resolution function [47]. Our test set contains references that are incorrectly resolved by the Eclipse, Netbeans, and IntelliJ editors and their refactoring tools. We even include tests that have given the Java compiler difficulties. Figure 2 shows a program that when compiled using javac 1.6.0_26, will result in a stack overflow[1]. We present this evidence, not as a critique of existing tools, but rather to underscore the complex nature of resolution. Specifically, we believe that when dealing with high-consequence applications, paying close attention to issues surrounding the correctness of resolution is justified.

III. Related Work

Research related to $\mathcal{M}onarch$ migration falls into several areas. One area of research focuses on changes to APIs and libraries as well as the propagation of these changes to client applications which use them. A second area of research focuses on streamlining or reducing runtime attributes of applications such as memory footprints and performance. It should be noted that $\mathcal{M}onarch$ migration is removal-based and thus a reduction in size of the source code of a migrated code base is a byproduct and not a goal of migration. That being said, the SCore classloader does perform specific optimizations on class files aimed at reducing the size of the executable code, and thus advances in reduction are of interest and relevant to SCore applications.

A. Evolution, Change, and Propagation

Bartolomei et al. [45] consider challenges relating to API migration. Whereas $\mathcal{M}onarch$ migration is removal-based and involves migrating source code implementations of class libraries to a specific platform, the API migration studied by Bartolomei et al. concerns itself with implementing the functionality provided by one API, called the *source API*, in terms of the functionality provided by another API, the *target API*. Under ideal circumstances, the interface of the source API would remain unchanged and only its implementation would shift. In practice achieving such a complete migration is not easy. In [45] the API migrations SwingWT and SWTSwing are considered. Five abstracted design patterns are identified as beneficial to API migration. A related work [3] explores API migration between the conceptually simpler and more similar XML APIs JDOM and XOM.

Tip et al. [43] describe how type constraints [32] can be used to correctly perform the EXTRACT INTERFACE refactoring, a kind of generalization. The goal of the EXTRACT INTERFACE refactoring, is to generalize an existing class by lifting an abstracted subset of the class to a newly created interface. The challenge then is to use this new interface type, in a maximal way, in the code base under consideration. Basic transformations involve substituting the type of a declaration with the interface type. Their work, implemented in Eclipse, performs source code analysis on a target code base. As is typically done in refactoring, the source code base being refactored is expected to satisfy the *closed world assumption*, meaning the refactoring tool has access to the program's full source code. In this context, type constraints are used to obtain the maximal set of program elements to be refactored.

[1]This bug has been fixed in Java 1.7.0_03.

Chow and Notkin [7] describe a transformation-based approach for maintaining compatibility between a library and its client applications. Their approach requires that changes to a library be captured, by library maintainer, in the form of annotations. Using information stored in these annotations, standard compiler technology can then be used by the application maintainer to generate transformations that make a given target application compatible with the new version of the library. It is noteworthy that application of transformations appears to rely heavily on syntactic properties and thus the complexity associated with semantic analysis is side-stepped. Conceptually, this form of transformation shares similarities with API migration. However, one distinction is that client applications are the subjects of modification. In contrast, $\mathcal{M}onarch$ migration is prohibited from modifying or even analyzing application code.

Dig and Johnson [15] have studied the role played by refactoring in API evolution. They explore incompatibilities that arise between an application and an API as a result of evolutionary changes to the API. The authors analyze four case studies and report that around 80% of API changes are refactorings that change the code structure of the API but do not change its behavior. These changes they classify as "non-breaking changes". In contrast, "breaking changes" are those that result in client code that fails to compile, or has altered behavior. They suggest that a refactoring-based migration tool could be used to perform most of the non-breaking changes needed to bring an application up to date with an evolving API upon which it depends. The remaining changes would need to be performed manually.

Henkel and Diwan [19] have developed a tool called *CATCHUP!* that has the ability to semi-automatically capture and replay refactorings. The purpose of the tool, whose prototype is implemented in Eclipse, is to capture refactoring-based changes that are applied to a library (i.e., a reusable software component) when it undergoes an evolutionary change and to then automatically replay the captured refactorings on application code bases that are dependent on the library. The authors mention that their tool is lightweight and their approach cost effective, though they also note that not all incompatibilities between a library and an application can be resolved through this mechanism.

Balaban et al. [1] explore the idea of class-to-class migration as a way to incrementally bring an application up to date with a new version of a library upon which it depends. The general approach assumes a one-to-one correspondence between legacy classes and replacement classes. Given such a correspondence, a set of rewrite rules can be manually created defining how references to the legacy class and its members are to be rewritten to corresponding references in the replacement class. Type constraints, such as those used in refactoring, are used to control the application of a given set of rewrite rules. Migrations from `Vector` to `ArrayList`, `Hashtable` to `HashMap`, and `Enumeration` to `Iterator` are used to illustrate their approach.

B. Reduction

The implementations of Java container classes are often targeted towards large use cases. For example, `Vector` and `Hashtable` are coded to have desirable performance properties in situations (i.e., operational profiles) in which the objects they store are heavily accessed. When this is not the case, these classes can be inefficient both in their execution speed and their memory footprint. Sutter et al. [41] investigate how profile information can justify the creation of custom classes whose functional behavior is an acceptable substitute for the original classes they replace. The purpose of customization is to reduce the memory footprint and improve execution speed of the application. The approach they present is fully automated and assumes type information about expressions and declarations is provided by the Java compiler. The remaining analysis is performed using *Gnosis*, a whole-program analysis and transformation tool developed at IBM. Within this framework, type-correctness constraints and interface-correctness constraints are expressed in rule-based format and provide information about requirements custom classes must satisfy (e.g., what declared elements they must contain) and where they can be used. The authors report that on the applications they considered, such customization resulted (on average) in a 22% speedup and 12% reduction of memory footprint. In contrast, $\mathcal{M}onarch$ migration involves a slightly different set of problems. First, our impression is that the nature and scope of customization

would present an unacceptable reliability risk. Second, $\mathcal{M}onarch$ cannot base its migration strategy on assumptions about the operational profile of application code. Third, the removal of code during $\mathcal{M}onarch$ migration is driven in part by "what if" type analysis whose information cannot be obtained from the compiler in a direct manner.

Mitchell et al. [28] analyze negative consequences of reuse and OO programming on large-scale Java applications. The authors use the term *runtime bloat* to refer to unnecessary time/space overheads accrued as a result of fairly standard development practices. The effects of runtime bloat become more pronounced when considering large-scale applications. In one instance cited, an application required 1 gigabyte of heap to service 500-1000 users – a level of resource usage whose scale to millions of users is prohibitive. In [29] the metric $(t - d)/t$ is developed, by Mitchel and Sevitsky, to measure the memory bloat factor associated with storing application data. In this formula t denotes the total number of bytes of live objects and d is the number of bytes corresponding to actual application data. Using this metric, Mitchell and Sevitsky report discover that a number of typical applications have a 60% to 80% memory overhead. A significant contributor to runtime bloat stems from reuse, which generally leads to increased programmer productivity with little consideration given to its impact on performance. Solutions proposed to prevent or limit runtime bloat include: (1) integration of suitable performance considerations into the design process, (2) development of tools capable of analyzing runtime bloat, (3) automated recognition and optimization of common bloat patterns, and (4) extension of the Java language to enable finer control over storage.

Tip et al. [44] explore the problem of reducing the size of applications that are distributed over the internet or used in embedded systems. Reduction techniques employed include cleanup-based "extraction techniques" that remove unreachable methods and redundant fields as well as transformations that perform method inlining, identifier compression, and class hierarchy compression. On average, applications were reduced to 37.5% of their original size. *Jax* is used to implement extraction and transformation techniques. The approach presented assumes, at least conceptually, that complete code base is available for analysis. When a complete code base is not available, assumptions falling outside the scope of static analysis can be expressed in the form of assertions using a modular extraction language (MEL). Different extraction scenarios are considered including those that create an extracted version of a library independently from any specific application. The core of the approach appears to center on the removal of elements from a class that are unreachable with respect to a given set of assumptions.

Rayside and Kontogiannis [35] describe how subsets of Java libraries can be extracted for use in embedded systems. The idea is to analyze how an application uses a library and to then filter out from the library all classes, methods, and fields that are not used by the application. This subset, they refer to as the "space optimized subset" of the library. Variations of this subset are also explored. Specifically, a "partial space optimized subset" can be constructed by strictly performing filtering at the level of compilation units (i.e., class files). The authors note that such subsets can only be guaranteed to function correctly for VMs that use lazy resolution. A "space reduced" subset is also presented suitable for VMs whose evaluation is strict. In contrast, $\mathcal{M}onarch$ cannot perform the filtering optimizations described in [35] because it is prohibited from accessing application code. However, the SCore processor uses a special classloader that performs various optimizations, including filtering out all methods from a library that are not used by the application. So in our framework, such filtering is a post-migration issue.

Pugh [34] explores how Java class files can be compressed in a wire-format in order to decrease the time it takes to transmit class files across, possibly slow, communication links. The assumption is that communication time is the dominating concern and class compression/decompression time is secondary. The wire-format compression techniques described result in files that are $1/2$ to $1/5$ the size of corresponding compressed jar files and roughly $1/4$ to $1/10$ the size of the original class files. Compression techniques include removal of high-level information found in LineNumberAttribute, LocalVariableTable, and SourceFile. However, these removals are not counted towards the final compression improvements. Main ideas in the compression that are counted include: (1) sharing Utf8 entries across class files (such entries often constitute the majority of the size of a class file), (2) partitioning of the constant pool into smaller "typed" constant pools (e.g., a separate pool just for method references, etc.) allowing pool

indexing to be reduced from 2 bytes to 1 byte in many cases, and (5) encoding strings.

IV. Platform

The **Java Micro Edition (ME)**[31] platform primarily targets devices belonging to the Internet of things. These devices, which include mobile phones and televisions, have varying degrees of internet connectivity. Through *configurations* Java runtime environments can be created satisfying a variety of criteria including memory footprints and processing power. In this context, a configuration consists of a VM and a set of class libraries. The Java ME supports two broad configuration types: (1) Connected Limited Device Configuration (CLDC), and (2) Connected Device Configuration (CDC). Through profiles (a set of higher-level APIs) and optional packages, the functionality of a configuration can be further customized to specific device categories. Connectivity implies that Java ME platforms must provide class loading, linking, and (simplified) bytecode verification capabilities. The Java ME platform also supports the full set of bytecodes. This makes the platform considerably larger (and more capable) than the Java Card platform.

The **Java Card**[30] platform encompasses a VM, a standard Java compiler, a converter (performing class loading, linking, and bytecode verification) and a defined set of APIs. Vendors can build their own smart card (hardware) implementing the VM of a Java Card platform – it is this hardware that is targeted by the converter. The hardware must also support/implement necessary functionality of the Java Card platform's APIs as well as any card-specific OS functionality. The result is an instance of the Java Card platform – programmers can now write *Java Card* programs (a subset of the *Java* Language) to be executed on this platform. A key domain targeted by the Java Card platform is one where a Java Card interacts with its environment through a card reader. Notable restrictions of the Java Card Language and Java Card VM include: no support for native methods, multidimensional arrays, char, long, and floating point operations. Bytecodes are also encoded differently than on the JVM in order to compress their size.

The **SCore** platform is a Java platform, developed at Sandia National Laboratories. This platform consists of a SCore-based VM, a standard Java compiler, a converter we will call interlude, and a (possibly evolving or customizable) subset of the Java SE APIs. In the SCore-VM the functionality of the selected SE APIs is obtained not through direct implementation but by migrating the source code implementations of the SE APIs to the SCore platform. This migration is accomplished using a transformation-based migration tool called Monarch.

A. Rationale and History of the SCore

It should be noted that the initial development efforts of the SCore predated the Java ME as well as any viable version of the Java Card. Initial investigations, which ultimately lead to the decision to develop the SCore, involved thorough review of publicly available information about Sun Java Card technology as well as in-depth meetings with well-known Java experts such as Bill Venners.

Although the SCore is suitable for use as a general purpose processor it's primary intent is for use in high consequence systems where programming errors could be catastrophic and where the security of the system is dependent upon knowing that there are no holes or back doors lurking in the shadows that could be exploited maliciously, or accidentally in the case of a fault somewhere else in the system. On the surface the SCore may appear to be a redundant implementation of technology available from commercial vendors. However, the SCore is in large part a product of the requirements placed on the systems for which it is intended and therefore, although it takes advantage of as much publicly available information as possible, many of those requirements make it extremely difficult to simply apply commercially available technology products in place of the SCore.

In order to eliminate the introduction of security holes, traps, or back doors through programming errors or by malicious programmers, unlimited access to ALL source and design information, both hardware and software, pertaining to the implementation must be available for analysis and inspection. In addition, personnel with a mastery of that information must be available to exhaustively evaluate and, if necessary,

to modify and/or correct any part of the implementation found to be incorrect or not fully supporting the security mandate of the system. Basically, any implementation must be explicitly shown to prevent either intended or unintended execution of a program fragment that circumvents implemented security measures or that violates the security barriers of the system in which the implementation resides. This includes any mechanisms that could cause data and/or program code to be accessed or modified in a manner not specifically required for the correct functioning of that implementation.

Some common examples of programming errors that can enable a security breach are unchecked buffer boundaries that provide a mechanism for overrunning said buffers, and pointer manipulation errors that allow data to be written or read from supposedly protected areas of memory. If the hardware doesn't preclude these breaches by design in a clear and understandable way that is provable by inspection by knowledgable, but Independent reviewers (or through other more rigorous methods), then the entire implementation must be analyzed, inspected, and reviewed in depth to ensure that no part of the implementation contains any intended or unintended avenues to circumvent the security of the system.

If, for example, the Java Micro Edition were to be run on a commercial microprocessor system that had previously been vetted and was deemed to meet the requirements of the intended system[2], one would still need to acquire and exhaustively inspect the Java Micro Edition source code, possibly for several different versions if one version was determined to contain faults that were fixed in a subsequent version, to ensure there were no hidden "holes" or "back doors" waiting to be exploited either accidentally or intentionally. In third party implementations like commercial microprocessors and software runtime environments like the Java Micro Edition, obtaining proprietary source code can be very expensive and nearly impossible to achieve in a timely manner, and finding and training personnel in the details of those implementations such that they can decisively determine the integrity of them is even more expensive in both time and money.

In contrast to that type of implementation, the SCore processor was designed from the ground up to exclude the kinds of errors described above and to prevent them from occurring. This protection is coded directly in the microcode of the processor and is easily analyzed and verified by knowledgable personnel[3]. The protection mechanisms run at hardware speeds increasing the overall performance of the system for a given set of environmental and power requirements, and, once these mechanisms have been verified, the security driven programming burden on the software developers is significantly reduced[4].

An important part of the justification for selecting a processor for use in a system application is how well it is supported with powerful and easy to use development and testing tools. Of the many capabilities the SCore development environment offers a development team two fundamentally important ones are the ability to quickly and easily model and simulate the desired system at varying levels of fidelity as the requirements and design are developed, implemented, and refined, and the ability to quickly and easily design and test custom I/O modules.

Simulation models of the SCore exist at various levels of fidelity, including a clock cycle accurate model that executes the exact same binary microcode and binary application code that the synthesize-

[2]Security, as well as power, environmental, etc. Environmental requirements could include things like radiation if the system was intended for space or nuclear applications.

[3]The SCore microcode is written in a high-level C-like syntax that is easily learned and can be mastered by any competent software engineer or programmer. Because it is a "hardware" language there are some restrictions and differences from traditional programming languages that must be understood and adhered to. Once these restrictions and differences are understood, reading and inspecting the microcode can proceed at a reasonable pace comparable to any other programming language. Because the microcode is designed and written in a high level language modern software development techniques are used to encapsulate the checking algorithms, and to reuse them for similar instruction implementations, making the analysis and inspection of these security measures easier. Instructions with common behaviors are grouped together so that the commonalities can be encapsulated to minimize cut-and-paste type errors as well, with mechanisms included in the language to allow inlining of the sequences of microcode instructions during translation for performance purposes where necessary.

[4]It should be noted that once any other JVM based implementation has been verified much of the same reduction in programming burden will be realized. However the SCore architecture goes further in protecting against programming errors and malicious programming than the traditional JVM. One example of this is the complete separation of address spaces (Program, Stack, and Heap/IO spaces), preventing the access of an illegal address space, either intentionally or inadvertently, during the execution of an application program.

able HDL[5] model executes. These models run on a Java based simulation environment that is used to model and simulate large, multi system designs where different elements in the system are modeled at differing levels of fidelity as their respective designs are refined and implemented. The SCore and it's array of available core I/O modules are included as built-in models with the simulation environment. The simulation environment, called Orchestra, has been used to model several large systems and is capable of modeling purely behavioral, as well as detailed logic designs like the SCore and it's I/O.

Besides system level modeling available in Orchestra, the SCore also includes fully functional Bus Functional Models (BFMs) to assist with SCore based I/O module design and testing, allowing the designer to concentrate on the I/O module design without having to build a complete SCore system and write Java code to test their design until the module is ready for certification and release. Identical BFM models exist for both Orchestra and for HDL simulation, and both BFMs execute the same script written in a Bus Functional Language (BFL) that is compiled into an internal structure providing a more efficient execution within the respective simulators. The BFMs provide a clock cycle accurate representation of the bus transactions on the SCore's Heap Bus for the custom Object Oriented I/O instructions of the SCore. The BFMs also provide an unlimited number of outputs and inputs that can be used to stimulate the I/O module-under-development's inputs and record it's outputs, respectively. The BFL also contains elements to help the developer document the functionality and the testing of a module.

B. The Scalable Core

The Scalable Core (SCore) platform [27][46] is a hardware implementation of the JVM [24] being designed at Sandia National Laboratories for use in resource-constrained embedded applications. The SCore has the following restrictions:

1) **prohibited use of floating point arithmetic** – Hardware implementations of floating point arithmetic are extremely complex and pose a serious threat to reliability.
2) **prohibited use of threading** – Concurrency based on multi-threading requires resources and creates the potential for a variety of failures resulting from deadlock, starvation, and race conditions. Furthermore, such failures can be hard to discover.
3) **limited support of native methods** – Java provides a mechanism, called the Java Native Interface (JNI), that enables non-Java methods to be referenced within a Java program. In the SCore, native methods can be implemented in microcode. However, due to resource constraints only a small number of such methods have been implemented.
4) **prohibited use of reflection** – Reflection requires information contained in class files, and utlizes native methods that are not supported on the SCore. The decision to not support reflection enables the SCore classloader to reduce the size of class files by excluding reflection-specific information.

An overview of the restrictions of the SCore processor is shown in Table I.

1) The SCore Classloader

As shown in Figure 1, development for the SCore begins on a standard desktop, running on a standard JVM. Embedded application developers may only use an agreed upon subset of the Java Core Libraries in their implementation. After the initial stage of development, the application is moved to a SCore-base discrete-event simulator, called *Orchestra* (not shown in the figure) [52], in which the system under development can be modeled. When development in the simulation stage is completed the application is then ready for execution on the SCore. Migration to a SCore-based environment involves (1) migration of the agreed upon subset of the Java Core Libraries, (2) compilation of the application code and the code of the migrated Libraries, and (3) translation of the resulting class files into a *ROM image* by a classloader-like program called $\mathcal{I}nterlude$. Using techniques similar to the *smart linking* mentioned by Reddy [36], and space optimization described by Rayside et al. [35], $\mathcal{I}nterlude$ combines all class files into a single significantly reduced file format called a *ROM image*. It is this ROM image that is executed by the SCore platform.

[5]Hardware Description Language. Currently the SCore exists as synthesize-able VHDL.

Java Language Restrictions

Feature	Keywords	Level of Support
floating point arithmetic	float, double, strictfp	not supported
threading	synchronized, volatile	not supported
serialization	transient	not supported
assertions	assert	not supported
multi-dimensional arrays		not supported

VM Restrictions

Feature	Keywords	Level of Support
reflection		not supported
native methods	native	limited
garbage collection		limited
class loading		not supported

Table I
RESTRICTIONS OF THE SCORE PROCESSOR

$\mathcal{I}nterlude$ enforces the requirement that an application only interacts with the migrated portion of the Java Core Libraries. If an embedded application contains references to elements (e.g., packages, types, methods, or fields) external to the portion of the Libraries migrated to the SCore, then $\mathcal{I}nterlude$ will not generate a ROM image. This property is relied upon by $\mathcal{M}onarch$[50] to assure the correctness of its migration.

V. CORE ANALYSIS

A Java code base consists of a set of packages, a package consists of a set of compilation units, and a compilation unit consists of an optional set of import statements and a set of top-level types. One thing to note is that in Java there are no special scoping rules for what might appear to be a sub-package. That being said, Java code is typically arranged in file hierarchies in a manner that might suggest otherwise. For example, `java.lang.annotation` is not a sub-package of `java.lang`.

In our discussion, we eliminate the structural aspects of Java code and model source-code as flat set whose elements are the *structural names* of top-level types. In our model, the *structural name* of a type has the form:

$$p \cdot cu \cdot t$$

where (1) p denotes the package in which the type resides, (2) cu denotes the compilation unit in which the type resides, and (3) t denotes the unqualified name (i.e., simple identifier) of the type. From the perspective of resolution, compilation units contain critical information. Primarily, compilation units contain import statements which allow references to elements declared in other packages. However, that is not all. Java's resolution algorithm requires that, within a compilation unit, types imported using single-type import statements take precedence over types, having the same name, that are declared in another compilation unit belonging to the package in which the compilation unit resides.

Aside from minor syntactic differences, the only difference between a structural name and a *canonical name*, as defined by Java, is the presence of the name of compilation unit.

$$p \cdot cu \cdot t \xrightarrow{\text{to canonical name}} p.t$$

The reason for modeling a code base in terms of the structural names of its top-level types is that it enables the standard semantics to be given to set operations (e.g., membership, subset, union and set difference) on code bases. Furthermore, given the mapping from structural to canonical names we extend set operations on code bases to include canonical names when the information provided by compilation units is not important. For example, let r denote the canonical name of a type and let $N = \{s_1, \ldots, s_n\}$ denote a set of structural names. The expression $r \in N$ can be evaluated by mapping all elements in

N to their canonical names and comparing results using the syntactic equality operation. The following table defines the functions *packages* and *canonical* which can be used to extract the set of packages and canonical names of top-level types from a code base.

$$\begin{array}{lcl} \text{packages(N)} & = & \{p \mid \exists p \cdot cu \cdot t \in N\} \\ \text{canonical(N)} & = & \{p.t \mid \exists p \cdot cu \cdot t \in N\} \\ \text{packageContents(P,N)} & = & \{p \cdot cu \cdot t \mid \exists p, cu, t : p \in P \wedge p \cdot cu \cdot t \in N\} \end{array}$$

Definition 1. *The symbol C_U denotes a target code base* derived *from a well-formed and complete code base U. Specifically, if C_U is derived from U, then $C_U \subseteq U$ holds.*

In the discussions that follow, we use the term *well-formed* to refer to code bases that are legal Java (e.g., they can be compiled into class files and executed). We use the term *complete* to capture the notion that a code base has not been altered from its original form/intent. For example, it is possible to remove some pieces of code from a well-formed and complete code base and obtain a well-formed code base. However, this new code base is not complete.

Note that, as a result of removals, C_U may contain references to elements whose declarations are in U but are external to C_U. To emphasize this distinction, we refer to U as a *complete code base* and a derived code base C_U as an *incomplete code base* or a *target code base*. Also note that, if given the actual source code corresponding to U the reification of C_U is unambiguous and straightforward. And finally, in practice a target code base represents the subset of a Library that embedded software developers wish to use.

The goal of the removal stage of $\mathcal{M}onarch$ migration is broadly stated as follows.

> *Migration Policy:* From the target code base, remove all fields, methods, and constructors having direct or indirect dependencies on (1) features that are not supported by the SCore, or (2) external references.

Unfortunately, when the analysis is exclusively limited to the target code base, a literal application of the above stated policy is not correctness-preserving for reasons that will be discussed in upcoming sections. This section lays the groundwork for such a discussion by introducing *resolution* which constitutes the foundation of the dependency analysis needed for removal.

It is important to know that, for assurance purposes (e.g., to facilitate manual code review and traceability of migration), $\mathcal{M}onarch$ operates exclusively on Java source code. In particular, $\mathcal{M}onarch$ may not extend its analysis to class files or jar files.

A. Resolution

We define a *reference* as a source-code expression (i.e., valid Java syntax) referring to a declared element. A reference is the mechanism by which types, arrays, fields, methods, constructors, can be denoted within Java source code. Such denotations can be in relative terms, in indirect terms (also known as aliases), and in absolute terms (also known as canonical forms). See Section 6.7 of the Java Language Specification for a discussion fully qualified and canonical names[18]. For reasons discussed shortly, we omit local variables and formal parameters from consideration.

On a conceptual level, a *reference* can be modeled as a (dot-separated) *sequence* consisting of one or more *atoms*, where an atom is either (1) a simple identifier denoting a package, type, generic type parameter, or field, or (2) an array reference, a method call, or a constructor call. This understanding of references as *sequences of atoms* leads to an incremental atom-based understanding of resolution. Atom sequences are resolved one atom at a time from left to right. We write $ref_{1..n}$ to denote a reference consisting of n atoms.

In this article, the term *resolution* is used to refer to a static analysis function that, relative to a code base C, maps element *references* to the canonical names of the declared elements to which they refer. We

use the term *resolvent* to denote to the result (i.e., the canonical name) produced by a resolution function. Let *ref* denote a reference and let t denote a canonical name of the type in which *ref* occurs. We formally express the resolution of (t, ref) relative to the code base C as follows.

$$resolution_C(t, ref) = resolvent \tag{1}$$

In this article, we restrict the domain of the resolution function under discussion so that resolvents produced (i.e., the range of the function) are the canonical names of types, arrays, fields, methods, or constructors. We use the terms *context* when referring to the source code location where a reference resides. As done in the Java Language Specification 6.5 [18], the term context allows us to make a finer distinction of the difference in perspective between references and declarations. A declaration has a *scope* (see 6.3 in [18]) and a reference occurs in a *context*. The goal of resolution is to determine in which declaration scope a reference falls. As stated in Section I-B, we also assume that $\mathcal{M}onarch$'s resolution function is *correct* in an ideal setting.

Axiom 1. *Let U denote a well-formed code base containing no external references. For such a code base, we assume $resolution_U$ is correct.*

On a more technical level, the resolution of a reference $(t_1, ref_{\,1..n})$ consisting of n atoms and occurring in the context type t_1 will, as shown below, involve $n-1$ resolution steps and n context types, $t_1, \ldots, t_n$.

$$(t_1, atom_1) \overset{step}{\Longrightarrow} (t_2, atom_2) \overset{step}{\longrightarrow} \cdots \overset{step}{\longrightarrow} (t_n, atom_n) \tag{2}$$

In practice, some minor adjustments need to be made in cases where the prefix of $ref_{\,1..n}$ denotes a package, but these details are unimportant for our discussion and the overall concept remains the same. The important points to keep in mind are (1) resolution of a reference consists of a *sequence* of resolution *steps* involving *individual* atoms, (2) the resolution of the first atom in a sequence uses a much broader search to find a resolvent than do subsequent resolution steps, (3) each resolution step has a corresponding *context type*, and (3) a context type is the canonical name of a type, which is obtained by full resolution of a type reference.

In figure 3, the resolution of reference `B.C.myD.myE.x` occurring in the context type `p1.A` involves the following context types:

$$p1.A, p1.B, p1.B.C, p1.D, p1.E$$

```
1  package p1;
2
3  public class A {
4          int x = B.C.myD.myE.x; // consider the reference: (p1.A, B.C.myD.myE.x)
5  }
6
7  class B {
8          static class C {
9                  static D myD;
10         }
11 }
12
13 class SuperD { E myE; }
14 class D extends SuperD {}
15 class E { int x; }
```

Figure 3. Example of a resolution sequence.

B. Local Variables and Generic Type Parameters

When viewed in its entirety, resolution must encompass references to local variables and generic type parameters. However, given the nature of the difference between complete and incomplete code bases we can, without loss of generality, restrict our discussion of resolution to contexts t denoting types. The reason for this is that the designation of a code base as being an complete code base or an incomplete code base does not effect how local variables or type variables are resolved. Therefore, we abstract away from our discussion resolution contexts associated with methods and constructors.

C. Restriction to Top-level Types

Without loss of generality, we restrict our discussion of resolution to references occurring in the context of top-level types. Note that target code bases C_U are (exclusively) obtained from complete code bases U by removing top-level types. No other kind of removal is considered or permitted. As a result, the contents of top-level types (e.g., member classes) are intact and any impact that information local to a top-level type (e.g., inner classes) has on resolution remains unaffected regardless of whether resolution is considered in the context of C_U or U.

D. Primary versus Secondary Resolvents

For the purposes of our analysis we will, when necessary, use the terms *primary resolvent* and *secondary resolvent* to make finer distinctions between resolvents. Such distinctions are relevant because the complexity of Java allows for the creation of code structures in which certain declarations are *hidden* (e.g., shadowed, overridden, or overloaded) from the context t in which the reference occurs. For a more detailed discussion of these scoping issues see the Java Language Specification [18] Section 6.4 (shadowing), Section 8.4.0 (method overloading), Section 8.4.8 (overriding and hiding), and Section 8.8.8 (constructor overloading). When they exist, we refer to such hidden declarations as *secondary resolvents* of the reference. Declarations that are not hidden are called *primary resolvents*.

A reference (t, ref) can have at most one primary resolvent. We designate the constant `<unresolved>` as the value that should be returned when an attempt is made to resolve a reference that does not have a primary resolvent (e.g., an external reference). We do not classify `<unresolved>` as a secondary resolvent.

A central concern is whether the migration process, when seen as a whole, creates conditions for the reclassification, during compilation, of a secondary resolvent as the (new) primary resolvent for a given reference. If such reclassification occurs, then migration is not correctness preserving. It should be noted that the incompleteness of target code bases as well as the removal of declarations that can occur during migration give rise to this threat.

A simple example highlighting a reclassification resulting from a removal is shown in Figure 4. It should be noted that, if the code in Figure 4 was selected for migration, $\mathcal{M}onarch$ would remove the both field declarations `B.x` and `B.y` so no reclassified references would exist in the migrated code. Nevertheless, the code shown in Figure 4 gives clear example of what is meant by *reclassification*.

```
1 package p;
2 class A { int x = 0; }
3
4 class B extends A {
5     double w = 2.0;
6     int x = (int) w; // this field will be removed during migration
7     int y = 5 / x;  // after removing p.B.x, the reference x will be resolved to p.A.x
8 }
```

Figure 4. Reclassification as a result of removal.

Within the context of $\mathcal{M}onarch$ migration, the removal of a declared element can occur in one of two ways, through tacit omission and explicit removal.

1) **Pre-migration**: Targeting a subset of a Library for migration can result in code bases for which there is a *tacit omission* of top-level type declarations. The goal of the preparation stage is to modify the target code base in order to eliminate the threats resulting from such tacit omissions.
2) **Migration**: If done incorrectly, the *explicit removal* of member declarations during migration can give rise to migrated code bases in which secondary resolvents are reclassified as primary resolvents. A discussion of how to eliminate this threat lies beyond the scope of this article, but is discussed in [50].

E. Unresolvable References

The element declarations that a Java code base can reference, either directly or indirectly, can be extensive. For example, let C denote a simple "hello world" program. Executing C via the command `java -verbose:class` reveals that this tiny program loads (i.e., has dependencies on) over 400 classes. What this means from the perspective of $\mathcal{M}onarch$ migration, is that if such a program is selected for migration the target code base[6] will either be (1) unreasonably large, or (2) contain *external references*. We call a reference an *external reference* if the element declaration to which it revolves lies outside of the code base C under consideration. Given the space limitations of the SCore, code bases targeted for migration always contain external references.

```
1  // ====================================================
2  package p3; // a subset of this package belongs to target code base
3  public class D {
4      public int x1 = p4.E.x;      // p4.E.x is an external reference
5      public int x2 = p4.E.myD.y; // p4.E.myD.y is an alias for p3.D.y
6                                  // whose prefix p4.E is an external reference
7      static int y = 1;
8      F myF2; // F should resolve to p3.F
9  }
10
11 // class F{} // suppose this top-level type is excluded from the target code base
12 // ====================================================
13 package p4; // this entire package is excluded from target code base
14
15 public class E {
16     public static int x = 0;
17     public static p3.D myD;
18 }
19
20 // ====================================================
21 package p5; // this package belongs to target code base
22 public class F {}
```

Figure 5. Some of the external references that need to be considered.

As defined in Chapter 6 of *The Java Language Specification*[18], Java's resolution algorithm involves a search of various artifacts and locations for resolvents to references. These artifacts include class files and jar files that can be located a various places on the desktop as well as URLs. The Java program is ill-formed if it contains a reference that cannot be resolved. In a well-formed code base, the *failure* to find a resolvent in one location initiates a search in another location. This continues until a resolvent is found. Noteworthy is the fact that failure to find a resolvent in one location (e.g., in a subtype hierarchy) does not imply a reference cannot be resolved.

[6]Recall $\mathcal{M}onarch$ only considers source code.

In contrast, when considering incomplete code bases, the failure to find a resolvent in one location means either (1) the resolvent resides in another location and the search should continue, or (2) as a result of a tacit omission, the resolvent is external to the code base being considered and the search should terminate. Therefore, when considering incomplete code bases, it is essential for a resolution function to be able to distinguish between these two possibilities. The complexity of resolution is such that care must be taken when making such a judgement. This is discussed in more detail in Section VI.

Let U denote a complete code base, and let C_U an incomplete code base derived from U. When resolving $(t, ref_{1..n}) \in C_U$, if $ref_{1..n}$ has a prefix that lies outside of the target code base C_U, then $(t, ref_{1..n})$ is classified as an *external reference*. The determination of whether a reference is external is formally defined by the predicate in Equation 3. Without loss of generality, the definition assumes that references $(t, ref_{1..n})$ explicitly incorporate the information from import statements – thereby eliminating the need for their consideration. For example, if the reference $(p1.A, B)$ resolves to a type that is imported via the single-type import statement $p2.B$, then the reference $(p1.A, B)$ is rewritten to $(p1.A, p2.B)$.

$$\forall (t, ref_{1..n}) \in C : externalReference_{C,U}(t, ref_{1..n}) \equiv \exists i : \begin{array}{l} 1 \leq i \leq n \\ \wedge \\ resolution_U(t, ref_{1..i}) = resolvent \\ \wedge \\ resolvent \notin C \end{array} \tag{3}$$

Figure 5 highlights some of the issues that must be considered when resolving references in target code bases. We close this section with an axiom stating a property of correct resolution within a complete code base U.

Axiom 2. *Let U denote a complete code base. The function $resolution_U$ never returns* `<unresolved>`.

$$\forall (t, ref) \in U : resolution_U(t, ref) \neq \texttt{<unresolved>}$$

VI. The Preparation Stage

We begin this section by introducing the definition of code composition and the two properties that drive preparation.

Definition 2. *The expression $C_1 \circ C_2$ denotes the composition of the code base C_1 with the code base C_2.*

$$C_1 \circ C_2 \stackrel{def}{=} C_1 \cup (C_1 - C_2)$$

The reason for not defining $\circ$ as the simple union of two code bases has to do with how such compositions are to be performed. Specifically, in such compositions types declared in C_1 should be given preference over same named types declared in C_2.

Completeness Property 1. *Given the code bases C_U and U. The function $resolution_{C_U \circ S}$ is* complete *if the following holds.*

$$\forall (t, ref) \in C_U : \begin{array}{l} \neg externalReference_{C_U,U}(t, ref) \\ \rightarrow \\ resolution_{C_U \circ S}(t, ref) \neq \texttt{<unresolved>} \end{array}$$

Consistency Property 1. *Given the code bases C_1, C_2, and C_3. The function $resolution_{C_1 \circ C_2}$ is* consistent *with $resolution_{C_3}$ if the following holds.*

$$\forall (t, ref) \in C_1 : \begin{array}{l} resolution_{C_1 \circ C_2}(t, ref) \neq resolution_{C_3}(t, ref) \\ \rightarrow \\ resolution_{C_1 \circ C_2}(t, ref) = \texttt{<unresolved>} \end{array}$$

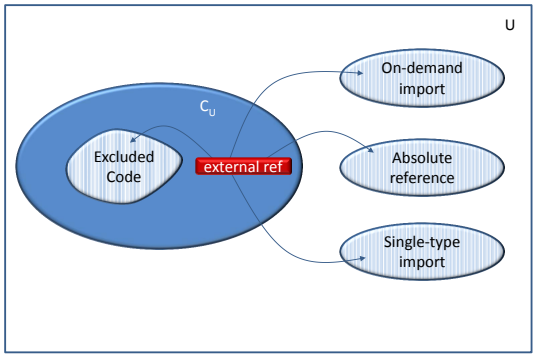

Figure 6. Sources of external references in C_U.

The goal of the preparation stage is to create a composition $C_U \circ S$, called a *prepared code base*, such that (1) $resolution_{C_U \circ S}$ is complete, and (2) $resolution_{C_U \circ S}$ is consistent with $resolution_U$. If these two properties are satisfied, the correctness of $resolution_{C \circ S}$ follows immediately from the correctness of $resolution_U$ (see axiom 1).

A. External Reference Sources

Let U denote a complete code base. As shown (in no particular order) in Figure 6, there are only four sources of external references in a derived code base C_U: (1) the reference refers to a declaration that has been excluded from a package belonging to C_U, (2) the resolution algorithm reaches a point where the reference is successfully matched with an on-demand import (static or non-static) statement that refers to a declaration that has been excluded from C_U, (3) the reference is an absolute reference to a declaration that lies outside C_U, or (4) the resolution algorithm reaches a point where the reference successfully matches a single-type import (static or non-static) statement occurring in a compilation unit in C_U that refers to a declaration excluded from C_U. In order for the resolution of references in C_U to be correct, it is necessary map such external references to the constant `<unresolved>`.

B. Code Skeletons

A *skeleton* is a code base in which bodies have been removed from methods and constructors, and initializations have been removed from fields. Note that skeleton types are similar to abstract types. A compilation unit is in skeletal form if all its types are in skeletal form.

Definition 3. *We use the symbol S_C to denote a skeleton constructed from the (entire) contents of the code base C.*

Though abstract, it is important to note that from the perspective of resolution, the declarations found in S_C are equivalent to the declarations found in C. For example, if a type t is declared in C then it is also declared in S_C. If a field f is declared in C it is also declared in S_C, as is the case for method and constructor declarations.

Theorem 1. *Let C denote a code base whose declared elements have unique names (i.e., unique simple identifiers). Let t_1 denote a type declared within C.*

$$\forall (t_1, ref_{1..n}) \in C : resolution_C(t_1, ref_{1..n}) = resolution_{S_C}(t_1, ref_{1..n})$$

Proof. (by contradiction) Let $ref_{1..i}$ denote the smallest prefix $ref_{1..n}$ such that $resolution_C(t_1, ref_{1..i}) \neq resolution_{S_C}(t_1, ref_{1..i})$. In this case, $resolution_{S_C}(t_1, ref_{1..i})$ = `<unresolved>`. This must be the case because S_C does not add any declarations to C and because the elements declared in C have unique

names, so no reclassification of secondary resolvents is possible. However for this inequality to hold, it must be the case that the resolution in C of the reference $(t_i, atom_i)$ found an element declaration (i.e., a resolvent) whereas the same resolution in S_C failed to find a corresponding element declaration. By definition of the construction of S_C this is not possible. □

Corollary 1. *Let C denote a code base and let t denote a type declared within C.*

$$\forall(t, ref_{1..n}) \in C : resolution_C(t, ref_{1..n}) = resolution_{S_C}(t, ref_{1..n})$$

C. The Supertype Closure Property

A significant portion of Java's resolution algorithm involves searching subtype hierarchies. To assure correctness, $\mathcal{M}onarch$ assumes that the target code base (not the prepared code base) is *supertype closed*. By this we mean that if a type t belongs to the target code base, then super type of t and all interfaces implemented by t must also belong to the target code base. Attempts to migrate a target code base that is not *supertype closed* will severely impact migration, and in the worst case will cause $\mathcal{M}onarch$ to produce an empty code base. There are several reasons for this. First, classes whose supertype definitions are not available are ill-formed and cannot be compiled. Second, resolution analysis frequently involves searching up subtype hierarchies. In high assurance settings, resolution must make conservative approximations when encountering a missing supertype. Such approximations lead to additional removals which can have cascading effects.

Property 1. *A target code base C is* supertype closed *if and only if it satisfies the following property.*

$$\frac{t_2 \in C \qquad t_2 <: t_1}{t_1 \in C} \textit{supertype closed} \tag{4}$$

D. The Package Closure Property

Packages may only be subjected to analysis in an all-or-nothing fashion. If only part of a package is targeted for migration, then this targeted code base must be composed with a skeleton containing the remainder of the package.

Property 2. *Let C_U denote a code base derived from the complete code base U. Let $P_{C_U} = packageContents(packages(C_U), U)$. The target code base C_U is* package closed *if and only if it satisfies the following property.*

$$\frac{t \in P_{C_U}}{t \in C_U} \textit{package closed} \tag{5}$$

The code shown in Figure 7 illustrates the need for package closure. Specifically, consider the situation where: (1) the class `p1.B` is omitted from the target code base, and (2) the class `p2.B` is included in the target code base. Under these conditions the resolution of the type of field `p1.A.B` will produce the (secondary) resolvent `p2.B` (instead of `p1.B`).

E. The On-demand Closure Property

$\mathcal{M}onarch$ has the the ability to automatically extract all on-demand import statements from a target code base. Given such a list, we assume that we can obtain the source code of all packages referenced by the extracted on-demand import statements.

Definition 4. *Let I_{C_U} be a package closed code base denoting the set of packages whose contents are accessible to the code base C_U via on-demand import declarations.*

```
1  // =========================================================
2  package p1;
3  import p2.*; // on-demand import of p2.B and p2.C
4  public class A {
5          B myB; // primary resolvent is p1.B
6          C myC; // primary resolvent is p2.C
7  }
8  // =========================================================
9  package p1;
10 public class B {}  // consider what happens if this
11                    // type declaration is omitted
12 // =========================================================
13 package p2;
14 public class B {}
15 // =========================================================
16 package p2;
17 public class C {}
```

Figure 7. The need for package closure.

Property 3. *Let C_U denote a code base derived from the complete code base U. The composition $C_U \circ S_{I_{C_U}}$ results in a code base that is* on-demand closed *with respect to C_U.*

It is important to note that `java.lang` will always belong to I_{C_U}, for convenience the on-demand import `import java.lang.*` is (tacitly) added by the compiler to all compilation units. The code shown in Figure 8 illustrates the need for on-demand closure. Specifically, consider the situation where: (1) the class `p2.B` is omitted from the target code base, and (2) the class `p1.MyC` is included in the target code base. Under these conditions the resolution of the `MyC` will produce the (secondary) resolvent `p1.MyC` (instead of `p2.B.MyC`).

```
1  // =========================================================
2  package p1;
3  import static p2.B.*; // resolvent changes if this line is omitted
4  public class A {
5          int x = MyC.x; // primary resolvent of MyC is p2.B.MyC
6                         // secondary resolvent of MyC is p1.MyC
7  }
8  // =========================================================
9  package p1;
10 public class MyC { public static int x = 2; }
11 // =========================================================
12 package p1;
13 public class C { public static int x = 1; }
14 // =========================================================
15 package p2;
16 // consider what happens if this type declaration is omitted
17 public class B { public static p1.C MyC; }
```

Figure 8. The need for on-demand closure.

F. The Prepared Code Base

Let C_U denote a code base derived from the complete code base U, that has been selected for migration to the SCore and is supertype closed. Let P_{C_U} denote the packages in C_U, and let $S_{P_{C_U}}$ denote the skeletal form of P_{C_U}. Let I_{C_U} denote the set of packages whose contents are accessible to C_U via on-demand import declarations, and let $S_{I_{C_U}}$ denote the skeleton of I_{C_U}.

Definition 5. *The prepared code base corresponding to C_U is defined as follows.*

$$Prep_{C_U} = C_U \circ S_{P_{C_U}} \circ S_{I_{C_U}}$$

On an operational level, it needs to be mentioned that $\mathcal{M}onarch$ adds special *skeleton* annotations to all element declarations within code skeletons. These annotations are retained during analysis. We introduce a function called $resolution'$ that is equivalent to $resolution$ with the following exceptions: (1) whenever a resolution sequence (as described in Section V-A) encounters a declared element that has a skeleton annotation, the function $resolution'$ terminates and returns `<unresolved>`, and (2) whenever an attempt is made to resolve a reference whose prefix is to a package external to the code base under consideration (e.g., the prepared code base), the function $resolution'$ terminates and returns `<unresolved>`. The following axioms formalize these assumptions.

Axiom 3. *Let U denote a complete code base, let C_U denote a code base derived from U, and let S denote a code skeleton corresponding to some subset of U.*

$$\begin{aligned}&\forall(t, ref) \in C_U : \\ &\quad externalReference_{C_U,U}(t, ref) \wedge \neg externalReference_{C_U \circ S,U}(t, ref) \\ &\quad \rightarrow \\ &\quad resolution'_{C_U \circ S}(t, ref) = \texttt{<unresolved>}\end{aligned}$$

Axiom 4. *Let U denote a complete code base, and let C_U denote a code base derived from U.*

$$\begin{aligned}&\forall(t, ref) \in C_U : \\ &\quad \neg externalReference_{C_U,U}(t, ref) \\ &\quad \rightarrow \\ &\quad resolution'_{C_U \circ S}(t, ref) = resolution_{C_U \circ S}(t, ref)\end{aligned}$$

Axiom 5. *Let U denote a complete code base, and let C_U denote a code base derived from U. Let $(t, ref_{1..n}) \in C_U$ denote a fully qualified reference. Let prefix $(t, ref_{1..i})$ where $1 \leq i < n$ denotes a package.*

$$\begin{aligned}&\forall(t, ref_{1..n}) \in C_U : \\ &\quad ref_{1..i} \in packages(U) \wedge ref_{1..i} \notin packages(C_U) \\ &\quad \rightarrow \\ &\quad resolution'_{C_U}(t, ref_{1..n}) = \texttt{<unresolved>}\end{aligned}$$

G. A Summary of $\mathcal{M}onarch$'s Resolution Analysis

A summary of the resolution algorithm that $\mathcal{M}onarch$ uses to resolve the *first atom* in a reference is shown in Figure 9. It is important to note that, in Java, the resolution of the first atom is treated differently from the resolution of all other atoms in a reference. In order to resolve the first atom in a reference a wide variety of searches are used, including searching encapsulating classes as well as import statements. After the first atom is resolved, all other atoms in the reference are resolved exclusively using a search up inheritance chains (i.e., supertype search). Since we require that target code bases are supertype closed, all resolution searches up the inheritance chain will, by assumption, produce correct resolvents. Thus, we focus our attention exclusively on issues surrounding the resolution of the first atom in a reference.

Without loss of generality, the summary in Figure 9 shows resolution in the case where the first atom is a simple identifier (and not a method/constructor call). It is important to keep in mind that this summary omits a number of technical details (e.g., nested classes, generic types, the impact of visibility modifiers, and method overloading). However, the summary does explicitly highlight places in the resolution algorithm where supertype closure, package closure, and on-demand closure are *necessary* for the correctness of resolution. The proof that these closures are necessary is "by example". Specifically, Sections VI-C,

1) Search 1: Interpret the first atom of the reference as a local variable
   a) Search frame.
   b) If this search fails, continue on to Search 2.
2) Search 2: Interpret the first atom of the reference as a field, method, or constructor.
   a) Search subtype hierarchy, beginning with the most closely encapsulating type t containing the reference.
      - Correctness assumption: supertype closed
   b) Search static imports (both single-type and on-demand).
      - Correctness action: on-demand closed
   c) If this search fails, continue on to Search 3.
3) Search 3: Interpret the first atom of the reference as a Type
   a) Search subtype hierarchy, beginning with the most closely encapsulating type t containing the reference.
      - Correctness assumption: supertype closed
   b) Search static imports (both single-type and on-demand).
      - Correctness action: on-demand closed
   c) Search non-static single-type imports for element.
   d) Search package contents (all compilation units within the package).
      - Correctness action: package closed
   e) Search non-static on-demand imports.
      - Correctness action: on-demand closed
   f) If this search fails, continue on to Search 4.
4) Search 4: Interpret prefix of reference as a Package
   a) Search for package.
      - Correctness action: none - search will fail if package is external
   b) If this search fails, the reference is `<unresolved>`.

Figure 9. A summary of $\mathcal{M}onarch$'s algorithm for resolving the first atom in a reference.

VI-D, and VI-E give relatively straightforward arguments or source-code examples demonstrating that resolution errors will occur if a particular closure property is not satisfied.

The argument that supertype, package, and on-demand closures are *sufficient* to assure resolution correctness is more involved. Without loss of generality we will restrict our discussion exclusively to top-level types. By this we mean that in all references (t, ref) both the t and ref are top-level types. The reason this is without loss of generality is that it is here where the problems arising from the analysis of incomplete code bases lie.

Lemma 1. *(Preservation of Primary Resolvents – the PPR lemma.) Let U denote a complete code base, and let C_U denote a target code base derived from U which is supertype closed. The removal of secondary resolvents from a C_U does not affect the ability of $resolution_{C_U}$ to locate primary resolvents.*

$$\begin{aligned}\forall(t, ref) \in C_U : \quad & \neg externalReference_{C_U,U}(t, ref) \\ & \rightarrow \\ & resolution'_{C_U}(t, ref) = resolution_U(t, ref)\end{aligned}$$

Proof. (by contradiction) Suppose $resolution_{C_U}(t, ref) \neq resolution_U(t, ref)$.

Case 1: $resolution_U(t, ref) \notin C_U$. In this case, $externalReference_{C_U,U}(t, ref)$, so the lemma is vacuously true.

Case 2: $resolution_U(t, ref) \in C_U$. Let C denote the largest code base such that $C_U \subseteq C \subseteq U$ for which $resolution'_C(t, ref) \neq resolution_U(t, ref)$. This implies there exists a single type $t' \in U - C$ whose inclusion in C would result in a code base $C' = C \cup \{t'\}$ such that $resolution'_{C'}(t, ref) = resolution_U(t, ref)$. Note that $t' \in U - C$ and $C_U \subseteq C$ implies $t' \notin C_U$. Because $\neg externalReference_{C_U,U}(t, ref)$ the type t' cannot explicitly participate in the resolution of (t, ref). The only other possibility for t' to affect resolution is for a search that was unsuccessful in C to become successful in C'. But this would require that t' be the result of the (now successful) search, in which case t' does explicitly participate in the resolution of (t, ref). But this implies $externalReference_{C_U,U}(t, ref)$.

□

Lemma 2. *Let U denote a complete code base, and let C_U be derived from U. Let the full contents of all packages in C_U be $P_{C_U} = packageContents(packages(C_U), U)$. Suppose $C_U \subset P_{C_U}$. What this means is that one or more top-level types have been omitted from packages whose top-level types are otherwise included in C_U.*

$$\begin{aligned}&\forall (t, ref) \in C_U : \\ &\quad \neg externalReference_{C_U \circ P_{C_U},U}(t, ref) \wedge resolution_U(t, ref) \in (P_{C_U} - C_U) \\ &\quad \rightarrow \\ &\quad resolution'_{C_U \circ S_{P_{C_U}}}(t, ref) = \texttt{<unresolved>}\end{aligned}$$

Proof. Since $\neg externalReference_{C_U \circ P_{C_U},U}$ we know, from the PPR-lemma that $resolution'_{C_U \circ P_{C_U}}(t, ref) = resolution_U(t, ref)$. From $resolution_U(t, ref) \in (P_{C_U} - C_U)$, corollary 1, and axiom 3 it follows that $resolution'_{C_U \circ S_{P_{C_U}}}(t, ref) = \texttt{<unresolved>}$. □

Lemma 3. *Resolution of external declarations accessible via on-demand imports. Let U denote a complete code base, let C_U be derived from U, and let $I_{C_U} \subset (U - C_U)$ denote the contents of all packages external to C_U that are accessible via on-demand imports, and let $S_{I_{C_U}}$ denote the skeletal form of I_{C_U}. Consider a reference $(t, ref) \in C_U$ whose resolution involves a successful match with an on-demand import statement in C_U.*

$$\begin{aligned}&\forall (t, ref) \in C_U : \\ &\quad \neg externalReference_{C_U \circ I_{C_U},U} \wedge resolution_U(t, ref) \in I_{C_U} \\ &\quad \rightarrow \\ &\quad resolution'_{C_U \circ S_{I_{C_U}}}(t, ref) = \texttt{<unresolved>}\end{aligned}$$

Proof. Note that on-demand import matching can only be used to resolve the *first* atom in a reference. Without loss of generality, let us assume that ref consists of a single atom whose resolution involves an on-demand import. Note that $\neg externalReference_{C_U \circ I_{C_U},U}(t, ref)$. Thus, from corollary 1 and the PPR-lemma it follows that $resolution_U(t, ref) = resolution_{C_U \circ S_{I_{C_U}}}(t, ref)$. From $resolution_U(t, ref) \in I_{C_U}$, corollary 1 and axiom 3, it follows that $resolution'_{C_U \circ S_{I_{C_U}}}(t, ref) = \texttt{<unresolved>}$. □

The WLOG assumption in the proof of lemma 3 warrants further discussion. In the general case, a reference consists of a sequence of atoms. During the resolution of such a sequence, there only specific places where resolution can involve on-demand imports. In particular, aside from the first resolution step (see Section V-A), it is only the in resolution of the first atom in a context type that on-demand imports can be considered.

Lemma 4. *Let U denote a complete code base and let C_U be derived from U. Let $(t, ref) \in C_U$ denote a reference whose resolution involves a fully qualified name or which involves a single-type import statement.*

Suppose the package referenced by either the fully qualified name or the single-type import is external to C_U.

$$\forall(t, ref_{1..n}) \in C_U : \\ resolution_U(t, ref_{1..i}) \in packages(U) \wedge resolution_U(t, ref_{1..i}) \notin packages(C_U) \\ \rightarrow \\ resolution'_{C_U}(t, ref) = \texttt{<unresolved>}$$

Proof. Follows from axiom 5. □

Theorem 2. *(Completeness of the Prepared Code Base.) Let* U *denote a complete code base and let* C_U *be derived from* U. *Let* $S = S_{P_{C_U}} \circ S_{I_{C_U}}$. *The function* $resolution_{C_U \circ S}$ *satisfies the completeness property stated in Section VI.*

Proof: By the PPR lemma we know that the resolution of references $(t, ref) \in C_U$ for which $\neg externalReference_{C_U}(t, ref)$ holds has the property that $resolution'_{C_U}(t, ref) = resolution_U(t, ref)$. We also know that $resolution_U$ does not output the value <unresolved>. What remains to be shown is that $resolution'_{C_U}(t, ref) = resolution_{C_U \circ S}(t, ref)$. More specifically, we need to show that, in the case where $resolution'_{C_U}(t, ref) \neq$ <unresolved>, adding secondary resolvents to C_U via the composition $C_U \circ S$ does not prevent the resolution function from finding the primary resolvent. We leave this last step to the reader.

Theorem 3. *(Consistency of the Prepared Code Base.) Let* U *denote a complete code base and let* C_U *be derived from* U. *Let* $S = S_{P_{C_U}} \circ S_{I_{C_U}}$. *The functions* $resolution_{C_U \circ S}$ *and* $resolution_U$ *satisfy the consistency property stated in Section VI.*

Proof. By case analysis.

Case 1: $externalReference_{C_U \circ S}(t, ref)$. In this case, ref is either a fully qualified name whose prefix references an external package, or the resolution of ref involves a single-type import referencing an external package. In this case, lemma 4 lets us conclude that $resolution_{C_U \circ S}(t, ref) =$ <unresolved>.

Case 2: $\neg externalReference_{C_U \circ S}(t, ref) \wedge externalReference_{C_U}(t, ref)$. By axiom 3, we conclude $resolution_{C_U \circ S}(t, ref) =$ <unresolved>.

Case 3: $\neg externalReference_{C_U \circ S}(t, ref) \wedge \neg externalReference_{C_U}(t, ref)$. From the PPR lemma we conclude that $resolution_{C_U \circ S}(t, ref) = resolution_U(t, ref)$.

□

H. An Operational Perspective

In $\mathcal{M}onarch$ the prepared code base $C_U \circ (S_{P_{C_U}} \cup S_{I_{C_U}})$ must be used as the basis for dependency analysis. The creation of $C_U \circ (S_{P_{C_U}} \cup S_{I_{C_U}})$ is not fully automated. However, $\mathcal{M}onarch$ does give assistance in its creation. Specifically, $\mathcal{M}onarch$ provides the following:

1) A transformation that extracts all on-demand import statements from a target code base.
2) A graph which can be viewed in order to visually confirm the subtype hierarchy of the target code base C_U is supertype closed. To view such graphs, a plugin has been developed for Cytoscape [12], an open source tool for visualizing complex networks, which can be launched from within $\mathcal{M}onarch$.
3) A capability for converting a code base into skeletal form.
4) A capability for storing a code skeleton.
5) A capability for composing a target code base with a stored code skeleton.

In practice, code preparation is a manual process assisted by $\mathcal{M}onarch$. After $(P_{C_U} \cup I_{C_U})$ has been created it is transformed into a skeleton and internally stored. When the target code base C_U is selected

for migration, a composition is automatically performed to produce the $C_U \circ (S_P \cup S_{I_{C_U}})$, which is then used for dependency analysis.

A $\mathcal{M}onarch$ migration is then performed as follows:

- Select the target code C_U to be migrated.
- Select the stored model $S_P \cup S_{I_{C_U}}$ to be used.
- Apply the migration transformation.

VII. Conclusion

Java-based embedded processors have restrictions that limit the extent to which Libraries can be utilized in embedded software. The two primary concerns are (1) the memory footprint entailed by Libraries can be prohibitive, and (2) the presence of language features or native methods in a Library that are not supported by the embedded processor. Diametrically opposed to this is the improvement in software quality that can be achieved in embedded software through utilization of the abstractions provided in Java Core libraries. An effective solution to this dilemma is to identify, perhaps in an application-specific fashion, the essential abstractions provided by a Library and to then migrate the selected Library subset to the embedded development environment.

From the perspective of reliability, security, and safety, the major challenge when performing such migrations lies in providing high-assurance that the functionality of the migrated code is correct with respect to its original form. In regards to this issue a major threat centers on understanding how the omission of element declarations affects the resolution of element references. This article describes how closure properties can be combined with code skeletons in order to create a source-code base of reasonable size for which the resolution of references is correct.

APPENDIX

The code base targeted for migration is a subset of Java SE 6 update 18. The selected subset consists of code belonging to the following packages:

	Target Subset Size	Total Package Size
`java.io`	2 files	84 files
`java.lang`	42 files	108 files
`java.util`	15 files	96 files

Some standard metrics for the targeted code base are shown in Table II .

Packages	=	3
LOC	=	23209
Compilation Units	=	59
Classes	=	48
Interfaces	=	20
Enums	=	0
Static Fields	=	263
Instance Fields	=	26
Static Methods	=	271
Instance Methods	=	254
Constructors	=	120
Static Initialization Blocks	=	7
Static Initialization Blocks LOC	=	42
Instance Initialization Blocks	=	0
Instance Initialization Blocks LOC	=	0
Method Cyclomatic Complexity		
Mean	=	4
Standard Deviation	=	6

Table II
SOME STANDARD METRICS FOR THE TARGETED CODE BASE

The location of all initialization blocks is shown in Table III.

Class		**Line**	**Type of Initialization**
java.lang.Byte	=	66	Basic cache initialization
java.lang.Character	=	2065	Basic cache initialization
java.lang.Integer	=	584	Cache initialization
java.lang.Long	=	533	Basic cache initialization
java.lang.Object	=	23	Calls native method `registerNatives()`
java.lang.Short	=	186	Basic cache initialization
java.lang.System	=	41	Calls native method `registerNatives()`

Table III
LOCATION OF INITIALIZATION BLOCKS

A. A Listing of the Files Included in Migration

A listing of the files (i.e., compilation units) selected for migration is shown in Table IV. It should be noted that in the java.lang.System class, three methods have been manually removed. The reason for this manual removal is that two of these methods contained an anonymous local class extending a class lying outside of the targeted code base. The third method made a call (the only such call in the targeted code base) to one of the deleted methods. In order to avoid the potential for a secondary resolvent to emerge (i.e., the call to the deleted method is resolved to another method in the code base), the third method was removed. All three manually removed methods were called nowhere else in the targeted code base (a property that our tool can automatically check).

java.io	
IOException.java	Serializable.java
java.lang	
AbstractStringBuilder.java	Integer.java
Appendable.java	Iterable.java
ArithmeticException.java	Long.java
ArrayIndexOutOfBoundsException.java	Math.java
ArrayStoreException.java	NegativeArraySizeException.java
AssertionError.java	NoSuchFieldException.java
Boolean.java	NoSuchMethodException.java
Byte.java	NullPointerException.java
Character.java	Number.java
CharSequence.java	NumberFormatException.java
ClassCastException.java	Object.java
ClassNotFoundException.java	RuntimeException.java
Cloneable.java	Short.java
CloneNotSupportedException.java	StackTraceElement.java
Comparable.java	String.java
Enum.java	StringBuilder.java
EnumConstantNotPresentException.java	StringIndexOutOfBoundsException.java
Error.java	System.java*
Exception.java	Throwable.java
IllegalArgumentException.java	UnsupportedOperationException.java
IllegalStateException.java	
IndexOutOfBoundsException.java	
java.util	
Collection.java	NoSuchElementException.java
Comparator.java	Observer.java
EmptyStackException.java	Queue.java
Enumeration.java	RandomAccess.java
Iterator.java	Set.java
List.java	SortedMap.java
ListIterator.java	SortedSet.java
Map.java	

Table IV
A LISTING OF THE TARGETED FILES.

$\mathcal{M}onarch$ has the capability of extracting the set imports used by a targeted code base. For the code base selected for migration the set of imports is shown in Table V. For the targeted code base, on-demand reference closure requires the following skeleton:

$$S_{IC_P} \stackrel{def}{=} \{\texttt{java.io}, \texttt{java.lang}, \texttt{java.util}\}$$

REFERENCES

[1] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of OOPSLA 2005*, pages 265–279, San Diego, California, United States, 2005. ACM.

[2] Michael Barr and Brian Frank. Java: Too Much for Your System? In *Embedded Systems Programming*. Embedded Systems Programming, May 1997.

[3] ThiagoTonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API Migration for Two XML APIs. In Mark van den Brand, Dragan GaŽevic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 42–61. Springer Berlin Heidelberg, 2010.

[4] Ira Baxter. private communication. Date: 2014-12-10.

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, February 2010.

[6] David Blaza. Shifting sands: Trends in embedded systems design. *Embedded*, May 2012.

java .io .*	java .util .Comparator
java .io .IOException	java .util .Formatter
java .io .InvalidObjectException	java .util .HashMap
java .io .ObjectInputStream	java .util .Iterator
java .io .ObjectStreamClass	java .util .Locale
java .io .ObjectStreamException	java .util .Map
java .io .ObjectStreamField	java .util .Properties
java .io .Serializable	java .util .PropertyPermission
java .io .UnsupportedEncodingException	java .util .Random
java .nio .channels .Channel	java .util .StringTokenizer
java .nio .channels .spi .SelectorProvider	java .util .regex .Matcher
java .nio .charset .Charset	java .util .regex .Pattern
java .security .AccessController	java .util .regex .PatternSyntaxException
java .security .AllPermission	sun .misc .FloatingDecimal
java .security .PrivilegedAction	sun .nio .ch .Interruptible
java .util .*	sun .reflect .Reflection
java .util .ArrayList	sun .reflect .annotation .AnnotationType
java .util .Arrays	sun .security .util .SecurityConstants

Table V
THE SET OF IMPORTS FOR THE TARGETED CODE BASE.

[7] Kingsum Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 359–368, Nov 1996.

[8] Francesco Cibai. private communication. Date: 2014-12-18.

[9] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer Berlin Heidelberg, 2005.

[10] Joseph Kiniry Cormac Flanagan and Patrice Chalin. private communication. Date: 2014-12-11.

[11] Coverity. Static Analysis Verification Engine(SAVE). http://www.coverity.com/products/coverity-save/. Accessed: 2014-12-10.

[12] Cytoscape. http://www.cytoscape.org/. Accessed: 2014-12-10.

[13] Semantic Designs. Java Parser (Front End). http://www.semanticdesigns.com/Products/FrontEnds/JavaFrontEnd.html. Accessed: 2014-12-10.

[14] Semantic Designs. Life After Parsing: Got My Grammar... uh, now what? http://www.semanticdesigns.com/Products/DMS/LifeAfterParsing.html. Accessed: 2014-12-10.

[15] Danny Dig and Ralph Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389 – 398. IEEE, 2005.

[16] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (Third Edition)*. Addison-Wesley, New York, USA, 2005.

[19] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, New York, NY, USA, 2005. ACM.

[20] Peter Henriksen. private communication. Date: 2014-12-10.

[21] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[22] IS2T. The five top reasons for using Java in embedded systems. *News*, October 2013.

[23] Konstantin Knizhnik and Cyrille Artho. Jlint. http://jlint.sourceforge.net/. Accessed: 2014-12-10.

[24] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, editors. *The Java Virtual Machine Specification (Java SE 7 Edition)*. Oracle, 2011.

[25] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2007.

[26] Francesco Logozzo and Manuel FÃd'hndrich. On the relative completeness of bytecode analysis versus source code analysis. In Laurie Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, 2008.

[27] James A. McCoy. An Embedded System For Safe, Secure And Reliable Execution of High Consequence Software. In *Proceedings of the 5^{th} IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 107–114. IEEE, 2000.

[28] N. Mitchell, E. Schonberg, and G. Sevitsky. Four Trends Leading to Java Runtime Bloat. *Software, IEEE*, 27(1):56–63, Jan 2010.

[29] Nick Mitchell and Gary Sevitsky. The Causes of Bloat, the Limits of Health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 245–260, New York, NY, USA, 2007. ACM.

[30] Oracle. Java card technology. http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html. Accessed: 2014.12.16.
[31] Oracle. Java Platform, Micro Edition (Java ME). urlhttp://www.oracle.com/technetwork/java/embedded/javame/index.html. Accessed: 2014.12.16.
[32] J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.
[33] 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.
[34] William Pugh. Compressing java class files. *SIGPLAN Not.*, 34(5):247–258, May 1999.
[35] Derek Rayside and Kostas Kontogiannis. Extracting Java Library Subsets for Deployment on Embedded Systems. *Science of Computer Programming*, 45(2-3):245–270, November-December 2002.
[36] David Reddy. Java technology is ready for embedded programming. *Hearst Electronic Products*, March 2012.
[37] David L. Ripps. Java for embedded systems: An introduction to using Java in embedded systems. *JavaWorld*, September 1996.
[38] Jon Ritzdorf. Globalization Issues with Medical Device Embedded Systems. In *Documentation & Training Life Sciences*, June 2008.
[39] N. Rutar, C.B. Almazan, and J.S. Foster. A Comparison of Bug Finding Tools for Java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, Nov 2004.
[40] Sun. The History of Java Technology. *Sun Developer Network*, c. 1995.
[41] Bjorn De Sutter, Frank Tip, and Julian Dolby. Customization of Java Library Classes using Type Constraints and Profile Information. In *Proceedings of ECOOP 2004*, pages 585 – 609, Oslo, Norway, 2004.
[42] EE Times. Embedded systems will be everywhere, expert predicts. *EE Times*, July 2005.
[43] Frank Tip, Adam Kiezun, and Dirk Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of OOPSLA 2003*, pages 13–26, Anaheim, California, USA, 2003.
[44] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical Extraction Techniques for Java. *ACM Trans. Program. Lang. Syst.*, 24(6):625–666, 2002.
[45] Thiago Tonelli, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010.
[46] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. Winter. The SSP: An Example of High-Assurance System Engineering. In *HASE 2004: The 8^{th} IEEE International Symposium on High Assurance Systems Engineering*, pages 167–177, Tampa, Florida, United States, 2004. IEEE.
[47] V. Winter, , C. Reinke, and J. Guerrero. Using Program Transformation, Annotation, and Reflection to Certify a Java Type Resolution Function. In *Proceedings of the 15^{th} IEEE International Symposium on High Assurance Systems Engineering (HASE)*, January 2014.
[48] V. Winter, J. Guerrero, A. James, and C. Reinke. Linking Syntactic and Semantic Models of Java Source Code within a Program Transformation System. In *Proceedings of the 14^{th} IEEE International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2012.
[49] V. Winter, J. Guerrero, C. Reinke, and J. Perry. Monarch: A High-Assurance Java-to-java (J2j) Source-code Migrator. In *Proceedings of the 13^{th} IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2011.
[50] V. Winter, J. Guerrero, C. Reinke, and J. Perry. Java Core API Migration: Challenges and Techniques. In *Proceedings of the 2014 International Conference on Software Engineering Research and Practice (SERP)*, July 2014.
[51] V. Winter, J. Perry, H. Siy, S. Srinivasan, B. Farkas, and J. McCoy. The Tyranny of the Vital Few: The Pareto Principle in Language Design. In *Journal of Software Engineering and Applications (JSEA)*, volume 4, pages 146 – 155, March 2011.
[52] V. Winter, H. Siy, J. McCoy, B. Farkas, G. Wickstrom, D. Demming, J. Perry, and S. Srinivasan. Incorporating Standard Java Libraries into the Design of Embedded Systems. In Ke Cai, editor, *Java in Academia and Research*. iConcept Press, 2011.

Victor Winter Victor Winter is an Associate Professor in the Department of Computer Science at the University of Nebraska at Omaha. His research interests include: program transformation, Java source-code analysis, language design, and high-assurance systems.

James A. McCoy James McCoy retired as a DMTS from Sandia National Laboratories in 2011 after 20+ years of work in computer security and information surety. He received a patent in 2013 for a secure processor architecture that is the foundation for the SCore Processor and was the technical lead for the team that produced RAD hard ASICs containing Score-based systems. With Dr. Victor Winter he co-developed a high-level, self-defining language for writing microcode called Paradigm that is architecture and processor independent.

Jonathan Guerrero Jonathan Guerrero is a graduate student at the University of Nebraska at Omaha pursuing an MS degree in Computer Science, with a concentration in Programming Languages. He has helped to develop Sextant: a tool for static analysis/visualization of Java source code, and for his thesis he is developing Cassandra: a tool for analysis/visualization of AspectJ code. In his free time he plays piano and attends martial arts classes.

Carl Reinke Carl Reinke recived in MS in Computer Science in December 2013 from the University of Nebraska at Omaha. He presently works at Sandia National Laboratories in Albuquerque New Mexico.

James T. Perry James Perry Received B.S. and M.S degrees from the University of Nebraska at Omaha (UNO) in 2009 and 2010, respectively. Prior to receiving his M.S., he worked as a Graduate Assistant for Victor Winter at UNO. Since 2011 he has been developing Surety Software as a Member of the Technical Staff at Sandia National Laboratories.