SAND2013-6169C

Java Core API Migration: Challenges and Techniques

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

***Abstract*—The task of developing Java-based applications for embedded systems can be greatly enhanced by providing developers access to Java's Core APIs such as `java.lang` and `java.util`. Oftentimes, platforms used in embedded systems are scaled back versions of the JVM. As a result, Core APIs must be *migrated* in order to be compatible with a particular platform. A significant portion of such migration centers around the removal of field, method, or constructor declarations.**

This paper describes the challenges and techniques associated with the automated removal-based modification of Java source code. Driving this research is the need to migrate selected Java Core APIs to an embedded platform called the SCore processor. This migration is being performed using a tool called $\mathcal{M}onarch$.

***Keywords*-source-code analysis; program transformation; code migration;**

I. Overview

$\mathcal{M}onarch$ is a Java source-code migration tool being developed at the University of Nebraska at Omaha to assist in migrating Java Core APIs to the *SCore platform*, a hardware implementation of the JVM [5] being designed at Sandia National Laboratories for use in embedded systems. The SCore is not a full-blown JVM and places various restrictions on the class files it can execute. For example, the SCore does not support floating point arithmetic. Therefore, the compilation of migrated code may not contain any floating point bytecodes (a more detailed discussion of the SCore is given in Section II). In a nutshell, the central problem that $\mathcal{M}onarch$ must confront is how to produce a (migrated) code base suitable for execution on the SCore.

At this time, the primary code base targeted for migration is a set of Core APIs belonging to the Standard Edition (SE) of the Java Platform. Specifically, a subset of Java SE 6 update 18 consisting of compilation units drawn from `java.io`, `java.lang`, and `java.util` shown in Table I has been targeted for migration. Some standard metrics for the targeted code base are also shown in Table II.

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

1) **re-implementation** – This manual stage involves the re-implementation of "must have" functionality within the Core APIs in order to remove unwanted dependencies. Re-implemented code fragments are encoded as program transformations which can then be automatically applied (or replayed). Further discussion of re-implementation lies beyond the scope of this paper. In this paper, we assume we are working with a target code base for which the re-implementation stage associated with migration has been completed.
2) **preparation** – This manual stage involves an expansion of the target code base with the goal of obtaining a *prepared code base* whose properties satisfy the preconditions of $\mathcal{M}onarch$'s static analysis system. Preparation is necessary to assure the correctness of static analysis. The specifics of the preparation stage also lie beyond the scope of this paper.

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

Table I
CODE BASE CURRENTLY TARGETED FOR MIGRATION.

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

3) **removal** – This fully automated stage, which is the focus of this paper, consists of the application of program transformations expressed as *conditional rewrite rules*. These transformations remove field, methods and constructor declarations having dependencies on features not supported by the SCore platform. A novel feature of $\mathcal{M}onarch$ transformations is that they enable the conditional portions of rewrite rules to include nontrivial semantic properties (e.g., resolution of references and dependency analysis).

In this paper, the term *resolution analysis* is used to refer to static analysis whose purpose is to determine the relation between *references* to types and type members (e.g., fields, methods, and constructors) and their *declarations*. Resolution analysis is central to the removal stage of migration and our discussion assumes that the source-code analysis capabilities of $\mathcal{M}onarch$ can correctly perform resolution analysis for target code bases that have been suitably *prepared*.

Contribution: This paper focuses on identifying and addressing correctness challenges arising from removal-based migration. The discussion and analysis takes into account features and properties of Java such as upcasting, shadowing, overriding, and overloading. Also included in the analysis are constraints imposed by the SCore.

Outline: The remainder of this paper is as follows. Section II gives an overview of the SCore platform. Section III describes resolution analysis. Section IV justifies and articulates a *migration policy* governing the removal of type members so that resolution analysis of the pre- and post-migration versions of the targeted code base is consistent. This consistency is sufficient to assure that the compilation of migrated code, by the Java compiler, is functionally correct and an informal argument is made to this effect. Section V takes an in-depth look at research activities having an intersection with the problem discussed in this paper, and Section VI concludes.

II. Background: The Scalable Core Platform

The Scalable Core (SCore) platform [6], [11] is a hardware implementation of the JVM [5] being designed at Sandia National Laboratories for use in resource-constrained embedded applications.

Table III gives a summary of the features not supported by the SCore. Noteworthy capabilities of the SCore include: (1) a restricted form of garbage collection, and (2) full support of class initialization.

A. Software Development

From the perspective of *process*, developing code for the SCore is essentially identical to developing code for the JVM. Programmers can develop and debug programs on a desktop using an IDE such as Eclipse or Netbeans. Standard tools such as unit testers can be used to validate aspects of the software. It is important to mention that at this stage of development, the application interacts with the original (i.e., un-migrated) Core APIs such as `java.lang` and `java.util`. After this initial stage, SCore development is moved to a simulation environment where the application interacts with the migrated Core APIs. See [12] for a detailed discussion of the simulation environment.

Java Language Restrictions

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

VM Restrictions

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

Table III
List of Java features not supported by the SCore

B. $\mathcal{I}nterlude$: The SCore Classloader

A SCore application is a Java program that is compiled using a standard Java compiler. Migrated Core APIs are also compiled in this fashion. The resulting class files are then processed, as shown in Figure 1, by a classloader-like converter called $\mathcal{I}nterlude$[1] which 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.

From the perspective of classfile compression, noteworthy properties of the ROM include:

- **monolithic constant pool** - A ROM image contains a single monolithic constant pool constructed by combining and optimizing the constant pools of all the classfiles in the application.
- **minimization of method inclusion** - The only methods contained in a ROM image are those that are actually used in the application. A caveat here concerns itself with the undecidability of *points-to* analysis. Specifically, in cases where method overriding is involved, $\mathcal{I}nterlude$ performs method inclusion using a conservative approximation.

$\mathcal{I}nterlude$ assumes the class files given to it are produced by trusted Java compilers. This assumption is an entailment

[1] $\mathcal{I}nterlude$ is not the official name of the converter used by the SCore development team. It is a term introduced by the authors to more concisely reference the part of the SCore tool set that performs classfile conversion.

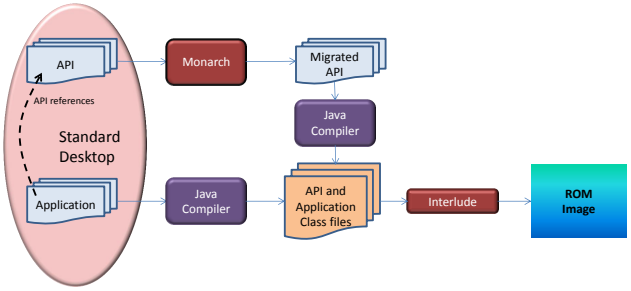

Figure 1. An overview of SCore application development.

of the development process for SCore applications which require all class files to be produced in-house from source code. Due to this assumption, general byte-code verification is unnecessary. However, $\mathcal{I}nterlude$ does verify that the class files it processes satisfy a specific set of properties. Among other things, $\mathcal{I}nterlude$ will fail to produce a ROM image upon encountering a symbolic reference to a non-existent field, method, constructor, type or package. The fact that $\mathcal{I}nterlude$ performs these important checks is relied upon by $\mathcal{M}onarch$ (see Section IV-A).

III. Core Analysis

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

> *Migration Policy 1:* From the targeted 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.

The removal stage is fully automated, transformation-based, and constitutes the heart of $\mathcal{M}onarch$ migration. For the remainder of this paper, we will use the term *migration* when referring to the removal stage of $\mathcal{M}onarch$ migration.

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, $\mathcal{M}onarch$ does not extend its resolution analysis to class files or jar files.

A. Resolution Analysis

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, and 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).

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 signature, or a constructor signature. This understanding of references as *sequences of atoms* leads to an approach to resolution analysis that is incremental in nature and atom-based: 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 paper, the term *resolution analysis* is used to refer to static analysis whose purpose is to determine the *relation* $\mathcal{R}$ between element *references* and element *declarations*. We use the term *resolvent* to refer to the result (i.e., the value) produced when resolution analysis is applied to a reference occurring in a given environment. 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) as follows.

$$resolution(T, ref) = resolvent \tag{1}$$

Note that in this paper, a resolvent is considered to be the canonical name of a type, array, field, method, or constructor.

1) Local Variables and Generic Type Parameters: When viewed in its entirety, resolution analysis must account for references to local variables as well as generic type parameters. $\mathcal{M}onarch$ performs such analysis during its migration. However, the goal of this paper is to create a source-code *removal policy* that will correctly remove declarations and all their corresponding references. Furthermore, the stated policy does not permit the isolated removal of local variables (e.g., from the bodies of methods) or generic type parameters. The only way local variables and generic type parameters could be removed during migration would be through the removal of their enclosing declarations (e.g., method declarations). It is precisely these enclosing declarations that define the scopes of local variables and generic type parameters. For this reason, it is without loss of generality that we can restrict our discussion of resolution analysis to environments (T) denoting types. In particular, we abstract away from our discussion the environments associated with methods and constructors.

B. 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* (i.e., not visible) from the environment (T) in which the reference occurs. When they exist, we refer to such hidden declarations as *secondary resolvents* of the reference. Declarations that are not hidden are called *primary resolvents*.

The Java compiler performs resolution analysis to determine reference bindings. $\mathcal{M}onarch$ performs resolution

analysis to perform dependency analysis used to determine which members of a type can be safely migrated, and which members must be removed.

Threat 1: A central concern is whether the migration process, when seen as a whole, creates conditions for the re-classification, by the Java compiler, of a secondary resolvent as the (new) primary resolvent for a given reference. If this shift occurs, then migration is not correctness preserving. It is the omission of declarations that gives rise to this threat. The omission of a declaration can occur in one of two ways.

1) **Pre-migration**: The consideration of code bases for which there is tacit omission of some declarations (e.g., targeting a subset of an API for migration). This situation is addressed during the preparation stage.
2) **Post-migration**: The explicit removal of declarations during migration.

C. Points-to *Analysis*

In theory, cases exist where resolution analysis is unable to precisely determine the primary resolvent of a reference. This particular problem is known as the *points-to* problem, and its static analysis is undecidable.

The undecidable nature of points-to analysis implies that an approximating analysis must be used to prevent errors associated with the potential re-classification of secondary resolvents. This approximation must be conservative with respect to correctness. From a technical standpoint, this means that cases can arise where $\mathcal{M}onarch$ may need to remove both primary and secondary resolvents in order to assure migration correctness. The drawback of such an aggressive removal policy is that it results in a (possibly unnecessary) reduction in the functionality that is migrated.

D. Unresolvable References

At the source code level, brute-force attempts to expand a target code base with the goal of obtaining a code base that is *reference-closed* are impractical. For example, a simple "hello world" program when executed via the command `java -verbose:class` demonstrates that this tiny program loads (e.g., has dependencies on) over 400 classes. Thus, when combined with the space limitations of the SCore, $\mathcal{M}onarch$'s restriction to source-code level analysis gives rise to target code bases containing *unresolvable references*. We denote such resolvents by the constant `<unresolved>`.

When resolving $(T, ref_{1..n})$, if $ref_{1..n}$ has a proper prefix that lies outside of the target code base C, then $(T, ref_{1..n})$ is classified as an *external reference*.

$$externalReference(C, (T, ref_{1..n})) \stackrel{def}{=} \exists i : \; 1 \leq i < n \wedge resolution(T, ref_{1..i}) = resolvent \wedge resolvent \notin C \tag{2}$$

From the perspective of migration, if an element in C has a dependency on an external reference, the element must be removed during migration. Thus, all external references can be treated as `<unresolved>`. Furthermore, for the purposes of dependency analysis, to conclude that a reference $(T, ref_{1..n})$ is `<unresolved>` it is sufficient to find a prefix $ref_{1..i}$ that is an external reference.

It should be noted that concluding that $(T, ref_{1..i})$ is an external reference is somewhat complex due to the nontrivial nature of resolution.

IV. Removal Analysis

This section identifies and articulates changes to the basic migration policy, stated in Section III, with the goal of producing an amended migration policy that is both implementable and correctness preserving. The central challenge here is to articulate a policy governing removal in such a way that the relation $\mathcal{R}$ between references and their primary resolvents is preserved in the migrated code.

Definition 1: Let C denote a target code base containing no unresolvable references (i.e., a compilable code base) and let C' denote the code base that results when C is migrated using $\mathcal{M}onarch$. Let $\mathcal{R}_C$ and $\mathcal{R}_{C'}$ denote the relations between references and their resolvents that exist in the code bases C and C' respectively. In order to be correct, removal-based migration must satisfy the following property:

$$migrationCorrect(C, C') \stackrel{def}{=} \mathcal{R}_{C'} \subseteq \mathcal{R}_C \tag{3}$$

It is worth noting that removing *more* (e.g., removing arbitrary elements) does not compromise correctness, provided this is done consistently and comprehensively. For example, the relation for an empty code base C' is $\mathcal{R}_{C'} = \emptyset$, and in this case $migrationCorrect(C, C')$ will hold for all C. This observation gives rise to an important secondary migration goal (whose discussion lies outside the scope of this paper). Namely, that migration should strive to maximize the amount of code migrated. This is after all, in part, what is being addressed in the re-implementation stage mentioned in Section I.

Policy Constraint 1: In order to assure migration correctness, $\mathcal{M}onarch$ may remove arbitrary declarations during migration. However, such removal should be kept to a minimum.

A. Application Independence

A constraint that has been placed upon $\mathcal{M}onarch$ is that API migration be *application independent*. For example, let $\mathcal{C}$ denote a code base (e.g., a subset of the Java libraries) that has been targeted for migration, and let *App* denote an application code base that uses the functionality provided by $\mathcal{C}$. Even though the analysis of *App* may result in the migration of a larger portion of C, $\mathcal{M}onarch$ may not broaden it static analysis to include *App*. We refer to this constraint as *application independence*.

There are technical as well as security related reasons justifying the application independence of API migration. On the technical side, if $\mathcal{M}onarch$ migration was application dependent, then $\mathcal{M}onarch$ would need to be intimately integrated with *all* (future) application development for the SCore. From the perspective of security, if an application is proprietary, then the confidentiality policy of an organization may prohibit "externally developed" tools such as $\mathcal{M}onarch$ from accessing the application.

> *Policy Constraint 2:* Migration of a targeted code base $\mathcal{C}$ must be performed in an application independent fashion. Specifically, migration may not make any assumptions about how an application might use $\mathcal{C}$.

As mentioned in Section II-A, application development for the SCore begins on a desktop environment utilizing a traditional JVM and only later transitions to the SCore platform. The application is also compiled with respect to this desktop environment. Migration must assure that applications developed and compiled on a standard JVM ultimately interact with migrated and *separately* compiled target code bases in a correct manner. From the perspective of migration, this approach to application development and separate compilation entails an additional challenge which is addressed by placing the following constraint on $\mathcal{I}nterlude$.

> *Policy Constraint 3:* The $\mathcal{I}nterlude$ classloader should fail to create a ROM image when the class files of an **application** contain a reference to a field, method or constructor that does not exist in the migrated target code base.

B. Shadowing

The following subsections examine the extent to which shadowing issues must be considered during migration.

1) Field Shadowing: In Java, it is possible for a subtype to (re)declare a field declared in its supertype. A re-declaration of a field x in a subtype T_2 (i.e., $T_2.x$) is said to *shadow* the declaration of the field x in its supertype T_1 (i.e., $T_1.x$). We also say the declaration $T_1.x$ is *shadowed* by the declaration in $T_2.x$, and that the declaration $T_1.x$ is *shadowing* the declaration $T_2.x$.

The semantics of field inheritance poses a threat to the correctness of field removal. For example, let (T, ref) denote a reference whose resolvent denotes the field re-declaration. In this case, the corresponding field declaration in the supertype constitutes a secondary resolvent. Thus, if the field re-declaration is removed, then the secondary resolvent will be re-classified as the primary resolvent for (T, ref). An example of this is shown in Figure 2.

```
1  package p1; // Target code base
2
3  public class A {
4      int y1 = new B2().x1;        // p1.B2.x1
5      int y2 = (int) new B2().x2; // p1.B2.x2
6      int y3 = (int) new B2().x3; // p1.B1.x3
7  }
8
9
10 class B1 {
11     int x1 = 1;
12     int x2 = 2;
13     int x3 = 3;
14 }
15
16 class B2 extends B1 {
17     // will be removed by Monarch
18     // shadows p1.B1.x1
19     int x1 = (int) 1.0;
20
21     // will be removed by Monarch
22     // shadows p1.B1.x2
23     double x2 = 2.0;
24 }
```

Figure 2. Field shadowing can threaten migration correctness.

> *Policy Constraint 4:* In order to assure the correctness of migration, when removing the field declaration $T.x$ it is also necessary to remove all field declarations shadowed by $T.x$.

2) Method Shadowing: Method shadowing is a phenomenon that only occurs between a nested class and its enclosing class(es). All references to shadowing methods must occur within the nested method where the shadowing method is declared.

Figure 3 gives an example of Java code in which the method declaration `p1.A.f()` is shadowed. This declaration is shadowed in the context of the class `InnerA`. In this case, the shadowing method declaration is `p1.A.InnerA.f()`.

In order to assure correct dependency analysis involving method shadowing it is necessary for a target code base to satisfy the *containment property* defined as follows.

Property 1: A target code base C satisfies the **containment property** if whenever a top-level class belongs to C, all its internals (e.g., nested classes) also belong to C.

The containment property is a natural property to assume for a targeted code base. Validating that a target code base

```java
package p1; // Target code base

public class A {
    public int f() { return 1; }
    public int g() { return 1; }


    public class InnerA {
        // shadowing method removed during
        // migration
        public int f() { return (int) 2.0; }

        // depends on p1.A.InnerA.f()
        public int y1 = f();
        public int y2 = g();
        public int y3 = z1;
    }

    public int z1 = 1;

    InnerA myThing = new InnerA();
    // int z2 = myThing.g(); // compile error
    // int z3 = myThing.z1;  // compile error
}

// ========================================
// Monarch may not subject this package
// to analysis
package app;

import p1.A;

class App {
        A.InnerA myThing = (new A()).new InnerA();
        int x1 = myThing.f();
        // int x2 = myThing.g(); // compile error

        int y1 = myThing.y1;
        int y2 = myThing.y2;
        // int y3 = myThing.z1; // compile error
}
```

Figure 3. An example of method shadowing.

```java
package p1; // Target code base

public class A extends B {
    // overrides p1.B.f()
    public int f() { return 1; }

    // overrides p1.B.g()
    // will be removed during migration
    public int g() { return (int)1.0; }
}

public class B {
    // overridden by p1.A.f()
    // will be removed during migration
    public int f() { return (int)2.0; }


    // overridden by p1.A.f()
    public int g() { return 2; }
}

// ========================================
// Monarch may not subject this package
// to analysis
package app;

import p1.*;

public class App {
        A myA = new A();
        int x1 = myA.f(); // p1.A.f()
        int x2 = myA.g(); // p1.A.g()

        B myB = new B();
        int x3 = myB.f(); // p1.B.f()
        int x4 = myB.g(); // p1.B.g()

        B myThing = new A();
        int x5 = myThing.f();
        int x6 = myThing.g();
}
```

Figure 4. An example of method overriding.

satisfies this property is part of the preparation stage of migration (which lies outside the scope of this paper). We explicitly mention containment property here for the sake of completeness, but will tacitly assume this property for the remainder of the paper. As a result, secondary-resolvent issues surrounding method shadows do not extend into the application. Thus, $\mathcal{I}nterlude$'s checks are sufficient to assure that method shadowing poses no additional challenges to migration, and therefore no specific policy need be developed to handle method shadowing (other than requiring the target code base to adhere to the containment assumption).

Policy Constraint 5: In order to be suitable for migration, a target code base C must satisfy the **containment assumption**.

C. Method Overriding

Method *overriding* occurs when a type T_2 declares a method m whose signature exactly matches that of a method declared in T_1 where $T_2 <: T_1$. In this case, we say the declaration $T_2.m$ *overrides* $T_1.m$.

Figure 4 shows a small class hierarchy in which method overriding occurs. In this case, `p1.B.f()` and `p1.B.g()` are *overridden methods*, and `p1.A.f()` and `p1.A.g()` are *overriding methods*.

Policy Constraint 6: When removing method $T.m$ all methods overridden by $T.m$ must also be removed.

1) Interfaces : It should be noted that the undecidability of points-to analysis provides sufficient justification for adopting a migration policy that deletes methods whose declarations are overridden by methods utilizing features unsupported by the SCore. However, this removal policy also solves the problem of method implementation requirements imposed by interfaces. Specifically, if a class implements an interface, then the class must have declarations for all abstract methods declared in the interface. Removal of such

an "interface method" from the class would result in a migrated code base that fails to compile.

D. Overloading

Method *overloading* occurs when methods have identical names but different signatures. A complete analysis of method overloading must take into account issues such as visibility, widening conversions, auto-boxing, and "closest fit" signature matching (which can get complex when a method signature contains multiple parameters).

From the perspective of migration, the case that must be considered is: "What happens if an application contains a reference to an overloaded method whose declaration resides in the original target code base, but whose declaration is removed as a result of migration?" In such a situation, secondary resolvents are possible. Furthermore, a particularly unfortunate form of overloading can arise involving parameters of type `java.lang.Object`. For example, Figure 5 reveals that the class `java.lang.StringBuilder` contains numerous overloaded declarations of its `append` method. Noteworthy is that (1) all declarations shown have the same arity, (2) the formal parameter of the first declaration is of type `java.lang.Object`, and (3) aside from a reference to the first declaration, a reference (in an application) to any other append declaration will have `append(Object obj)` as a secondary resolvent. The root cause of such secondary resolvents results not from overriding, but from how Java resolves references to overloaded methods.

```
1 public StringBuilder append(Object obj) ...
2 public StringBuilder append(String str) ...
3 ...
4 public StringBuilder append(float f) ...
5 public StringBuilder append(double d) ...
```

Figure 5. Overloaded declarations of the method `append` in the class `java.lang.StringBuilder`.

For references occurring within the target code base, secondary resolvents arising from overloading poses no analysis problems. However, this is not the case for references to overloaded methods occurring within the application, since these are not subjected to analysis. Particularly unappealing is the idea of expanding the removal of a declaration like `append(float f)` to also include the removal of `append(Object obj)` in order to assure the correctness of reference resolution within an application. Fortunately, the properties of $\mathcal{I}nterlude$ can be leveraged to avoid this situation. Specifically, $\mathcal{I}nterlude$ will fail to compile class files containing symbolic references to non-existent methods.

> *Policy Constraint 7:* Secondary resolvents resulting from references to overloaded methods/constructors impose no additional constraints on migration because $\mathcal{I}nterlude$ will fail to produce a ROM image if it encounters a reference to a non-existent method or constructor.

E. Special Cases

In the following sections discuss removal cases requiring specialized analysis.

1) Constructor Removal: $\mathcal{M}onarch$ migration exclusively consists of removal of fields, methods, and constructors – and anonymous types. Removal of named types is not supported since this has the potential to decimate a code base – an example of which will be discussed shortly. However, this restriction can pose a threat to the correctness of migration in the case when the constructors of a type are exhaustively removed. In particular, when encountering a class that does not explicitly contain a constructor, the Java compiler will automatically generate a no-argument default constructor for that class.

> *Policy Constraint 8:* Migration may not exhaustively remove all explicitly declared constructors belonging to a type.

We encountered the problem of exhaustive constructor removal in practice. In particular, the core API targeted for migration to the SCore containes the class `java.lang.Throwable`. Every constructor of `Throwable` had a dependency on a native method called `fillInStackTrace`. This native method is not supported on the SCore platform. Thus, the constructors for the class `Throwable` needed to be either exhaustively removed or or some subset needed to be re-implemented. Exhaustive removal of the `Throwable` constructors yielded unacceptable results since it mandated the (manual) removal of the class `Throwable`. This lead to a cascading sequence of removals that decimated the migrated code base. Specifically, since `Throwable` is the supertype of all `Exception` and `Error` classes, and since the `Exception` and `Error` classes were part of the code base targeted for migration, the removal of `Throwable` would also require the removal of the `Exception` and `Error` classes as well as any of their subtypes.

In this case, the dependency problem was resolved during the re-implementation stage (see Section I).

2) Initialization Blocks: A *static initialization block* is a mechanism that enables nontrivial initialization of static fields. Such an initialization capability is useful since class initialization methods (i.e.,g `clinit`) cannot be explicitly defined. In Java 1.1, anonymous classes were introduced and *instance initialization blocks* were introduced to compensate

for the fact that constructors may not be defined for anonymous classes.

When a static initialization block is present in a class, it is implicitly associated with the class initialization method for that class. Similarly, all instance initialization blocks in a class are associated with *all* constructors for that class. This makes the removal of initialization blocks problematic. For example, removal of an instance initialization block implies removal of all constructors for the class. Therefore, $\mathcal{M}onarch$ classifies initialization blocks as "must have" functionality whose dependencies must be addressed in the manual re-implementation stage (see Section I).

In practice, the use of instance initialization blocks is rare and the use of static initialization blocks is infrequent. In the Core API targeted for migration there are 7 static initialization blocks with average LOC = 6, and 0 instance initialization blocks.

Policy Constraint 9: The removal phase of migration classifies initialization blocks as "must have" functionality. Furthermore, $\mathcal{M}onarch$ will migrate all initialization blocks without performing dependency analysis. It is left to the re-implementation phase to assure that all initialization blocks are free from unwanted dependencies.

V. Related Work

Behavior-preserving refactoring [3] and removal-based migration both center around modifications to source code constrained by properties involving primary and secondary resolvents. Many standard refactorings involve either changing a declaration (e.g., renaming) or repositioning a declaration within a subtype hierarchy (e.g., extracting a superclass or converting a local variable to a field). For example, the *rename method* refactoring fully takes into account overriding and considers overloading issues to a limited extent. This is similar, but not identical to what is needed for $\mathcal{M}onarch$ migration. Noteworthy is that method renaming extends across the entire subtype hierarchy renaming method declarations in all sibling, cousin, and descendent classes. This is somewhat different from what is done in $\mathcal{M}onarch$ migration. Specifically, in order to maximize the number of declarations in the migrated code base, removal of a method only impacts ancestors (i.e., supertypes) of the class containing the method declaration flagged for removal and not its sibling, cousin, or descendent classes.

Refactoring has also been used to capture developers' manual modifications of libraries with the intent of replaying the captured actions on client software that uses the modified libraries [4]. In this case, a separation is made between the evolution of an API and modifications to applications using the evolved API. The goal of replay is to bring application programs "up to date" with the evolving APIs they depend on. Henkel and Diwan have developed a semi-automated refactoring capture-and-replay tool called CatchUp! for this purpose. Such replay abilities are similar to those in $\mathcal{M}onarch$ which formulate re-implementations as replayable transformations.

Refactoring with type constraints has been used to remove deprecated code [1] and convert legacy code with unchecked downcasts into type-safe generic code [2]. In a similar vein, a tool called *Rosemari* [10] has been developed to upgrade legacy applications making them compliant with evolving frameworks such as JUnit. In this case, code migration is based on annotation refactorings. For example, JUnit version 3 requires that test cases and test suites adhere to certain naming conventions. In contrast, JUnit version 4 imposes no naming conventions, but does require test cases and test suites to be properly annotated.

Our work differs from refactoring-based work related to Java libraries in that we exclusively modify the libraries not the client software that uses the libraries. In addition, whereas others focus on preserving the API exposed by the libraries, we reduce the functionality of the libraries because of restrictions of the target platform. To the best of our knowledge this has not been done elsewhere. Even Oracle does not provide a migrated version of the standard libraries for use on their Java Card platform [8].

Rayside and Kontogiannis [9] discuss a process to extract Java library subsets for supporting embedded systems applications by removing unused components from the library. They have the capability to produce library subsets having certain properties: (1) a space optimized subset, (2) a partial space optimized subset, and (3) a space reduced subset. The production of a subset is application specific with the space optimized subset being the most aggressive. The space optimized subset is created by removing all fields and methods that are not referenced by a given application. This is slightly different than the migration goals we are pursuing in which we want to universally prohibit access to fields and methods depending on features that are not supported by the target platform (i.e., the SCore). Furthermore, $\mathcal{I}nterlude$, the class loader for the SCore [7], has similar removal capabilities to the space optimized subset produced by Rayside and Kontogiannis. In particular, when processing the class files for a given application the class loader for the SCore removes all methods (but not fields) that are not referenced.

VI. Conclusion

This paper conducts an in-depth analysis of the impact that declaration removal can have on Java code bases. This analysis is performed in the context of an API migration effort whose goal is to produce Java source code suitable for execution on the SCore platform.

At this time, the primary code base targeted for migration is a set of Core APIs belonging to the Standard Edition (SE) of the Java Platform. Key constraints imposed on migration

are: (1) API removal analysis may not make assumptions about (SCore) applications using the migrated API, and (2) SCore application starts on a desktop environment containing the un-migrated form of the target Core APIs and finishes on a simulation environment containing the migrated version of the Core APIs.

From our experiences we conclude that Core API migration is possible and can be automated to a significant extent. A beta-version of this policy has been implemented in a migration tool called $\mathcal{M}onarch$, capable of fully automatic removal-based migration. The policy is enforced by both the $\mathcal{M}onarch$ migrator and the $\mathcal{I}nterlude$ classloader.

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] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java Programs to Use Generic Libraries. In *Proceedings of OOPSLA 2004*, pages 15 – 34, Vancouver, BC, Canada, 2004.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] 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)*, St.Louis, Missouri, USA, 2005.

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

[6] 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.

[7] Steve Morrison. SSP Class Loader Responsibilities. Technical report, Sandia National Laboratories, 2005. Internal Report.

[8] Oracle Sun Developer Network (SDN). Java card technology. http://java.sun.com/products/javacard/.

[9] 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.

[10] Wesley Tansey and Eli Tilevich. Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications. *SIGPLAN Not.*, 43(10):295–312, October 2008.

[11] G. L. Wickstrom, J. Davis, S. E. Morrison, S. Roach, and V. L. 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.

[12] Victor L. Winter, Harvey Siy, James McCoy, Ben Farkas, Greg Wickstrom, Doug Demming, James Perry, and Satish Srinivasan. Incorporating Standard Java Libraries into the Design of Embedded Systems. In Ke Cai, editor, *Java in Academia and Research*. iConcept Press, 2011.