Incorporating Standard Java Libraries into the Design of Embedded Systems

Victor Winter, Harvey Siy
Department of Computer Science
University of Nebraska at Omaha, USA

James McCoy, Ben Farkas, Greg Wickstrom, Doug Deming
Sandia National Laboratories, USA

James Perry, Satish Srinivasan
Department of Computer Science
University of Nebraska at Omaha, USA

1 Introduction

When used in embedded systems, JVMs are typically restricted in a variety of ways. The practical consequence to the Java developer is that not all Java programs can be executed on a restricted JVM. A notable example is that standard class libraries available to the “Java desktop programmer” may not be available or may only be partially available to the “Java embedded systems programmer”. This limitation can have far reaching consequences and can ultimately impact the overall design of an embedded system.

This chapter describes an ongoing effort to develop a highly-automated approach for migrating Java source-code, such as standard class libraries, to restricted JVMs. We are developing a transformation-based tool called Monarch\(^2\) capable of migrating Java source-code to restricted JVMs.

1.1 Background and Motivation

Despite the widespread use of desktop and laptop computers, most microprocessors and micro-controllers produced today are, in fact, used in embedded systems. In 2002, the EETimes reported that about 98% of all 32-bit processors produced were being used in embedded systems (Turley, 2002). Out of the nine billion processors manufactured in 2005, 8.8 billion were used in embedded systems (Barr, 2005). Furthermore, the micro market is growing. According to World Semiconductor Trade Statics (WSTS) forecasts, the

---

1This work was in part supported by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

2Butterflies are the archetype of transformation. Monarch butterflies are known for their migratory prowess, travelling roughly 2500 miles during their migration.
world’s semiconductor market is expected to top $320.2 billion in 2012, and the micro market is projected to top $68.8 billion.

1.1.1 Embedded Systems

An embedded system distinguishes itself from a general-purpose computing system, such as a desktop computer, in that it is designed to perform a specific set of functions (Heath, 2002). The environments in which embedded systems are expected to operate also typically impose a number of constraints on processors including (1) size constraints, (2) power consumption constraints, and (3) real-time constraints.

A common approach taken when developing a computationally sophisticated embedded system is to adapt a general-purpose computing platform such as the Java Virtual Machine (JVM). However in order to satisfy environmental constraints, the functionality of such general-purpose computing platforms must oftentimes be restricted or otherwise altered.

Example 1. The unpredictable response times associated with garbage collection can introduce delays into mission critical functions within an embedded system. Such delays may prevent the system from meeting its (hard) real-time requirements. The Java Real-Time System (RTS) is an adaptation of the JVM for use in systems having real-time requirements. In particular, a notable difference between the Real-Time VM and the JVM is the introduction of a new kind of thread which cannot be interrupted by garbage collection.

1.1.2 Adapting the JVM to Embedded Systems

There are several reasons motivating the adaptation of general-purpose platforms such as the JVM to embedded systems. First of all, a lot of effort has been put into the design of the JVM and its design has been subjected to substantial analysis both from industry as well as academia. In this respect, it can be argued that the JVM represents a highly mature computing platform. When designing an embedded processor, it is generally cost effective to leverage this maturity of the JVM to the fullest extent possible.

A second important reason for adapting the JVM (or some other general-purpose platform) to an embedded environment is based on the premise that mainstream programming languages and development tools such as IDE’s, debuggers, and unit testers will then be compatible with (at least) part of the development cycle. This is important because the use of mainstream programming languages and development tools is extremely cost-effective. Their design and implementation represents the culmination of a tremendous amount of development effort and every attempt should be made to leverage their capabilities when developing a system. In addition, these tools and languages have been heavily vetted by an extremely large and diverse user base and thus also represent mature technologies.

1.2 Chapter Overview

The remainder of this chapter is organized as follows: Sections 2 and 3 respectively give an overview of an embedded processor and simulation environment being developed at Sandia National Laboratories. It is in this context that Java code migration is being considered. Sections 4 and 5 are the centerpiece of this chapter and describe the migration problem, its challenges and our approach to addressing these challenges. Section 6 summarizes various related work, and Section 7 concludes the chapter.
2 The Scalable Core Platform

The Scalable Core (SCORE) platform (McCoy, 2000), (Wickstrom et al., 2004) is a hardware implementation of the JVM (Lindholm & Yellin, 1999) being designed at Sandia National Laboratories for use in resource-constrained embedded applications.

A SCORE application is a Java program that can be compiled using a standard Java compiler. The resulting class files are then processed, as shown in Figure 1, by a classloader-like translator called Interlude which combines all class files into a single significantly reduced file format called a ROM image. The ROM image is executable by the SCORE and contains a single monolithic constant pool. Interlude also assures that the only methods contained in the ROM image are those that are actually used in the application.

From a functional standpoint, the SCORE is a restricted implementation of the JVM because it does not support the entire set of Java bytecodes. Restrictions imposed by the SCORE platform include:

1. **prohibited use of floating point arithmetic** – The implementation of floating point arithmetic in hardware is a well-known source of errors. Thus, the presence of hardware implementing full floating point arithmetic would introduce a serious assurance burden for the SCORE.

2. **prohibited use of threading** – Multi-threading introduces additional resource and assurance burdens. First, at least conceptually, individual threads require separation of their execution space to ensure coherence. Both physical memory and required overhead for thread separation is expensive. Second, concurrent execution introduces a variety of semantic problems that can occur such as deadlock, where multiple threads are waiting for each other to release a desired resource.

In the framework being developed, the needs of real-time applications requiring periodic, aperiodic, and sporadic tasks are addressed within a cooperatively multi-tasked system. In such a system design, support for threading becomes unnecessary, and latency issues arising within such a non-preemptive system are mitigated through the use hardware-based queuing and distributed support infrastructures (Wickstrom, 2000).

3. **limited support of native methods** – The Java Native Interface (JNI) is the mechanism by which non-Java methods (aka native methods) may be called from within a Java program. For example, if an application needs to interface with special hardware devices, then a native method may be needed. The SCORE does not impose any theoretical limitations on the use of native methods. However, the SCORE platform currently only supports a subset of the native methods used in the standard
### Java Language Restrictions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating point</td>
<td>strictfp, float, double</td>
<td>unsupported</td>
</tr>
<tr>
<td>threading</td>
<td>synchronized, volatile transient</td>
<td>unsupported</td>
</tr>
<tr>
<td>serialization</td>
<td>transient assert</td>
<td>unsupported</td>
</tr>
<tr>
<td>assertions</td>
<td></td>
<td>unsupported</td>
</tr>
<tr>
<td>multi-dimensional arrays</td>
<td></td>
<td>unsupported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VM Restrictions</th>
<th>Relevant Keywords</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>native methods</td>
<td>native</td>
<td>limited support</td>
</tr>
<tr>
<td>garbage collection</td>
<td></td>
<td>limited support</td>
</tr>
<tr>
<td>reflection</td>
<td></td>
<td>unsupported</td>
</tr>
<tr>
<td>(dynamic) class loading</td>
<td></td>
<td>unsupported</td>
</tr>
</tbody>
</table>

Table 1: List of Java features not supported by the SCORE

Java libraries. In the current design of the SCORE, only five general-purpose native methods are supported: `exit`, `halt`, `gc`, `println`, and `cloneObject`.

4. **prohibited use of reflection** – In addition to having dependencies on (unsupported) native methods, reflection makes use of various information present within class files. Much of this information is excluded during the class file to ROM-image translation performed by Interlude.

Table 1 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) class initialization.

#### 2.1 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. Tools such as unit testers can be used to validate aspects of the software.

From the perspective of the *Java language*, software developed for the SCORE may not contain any of the unsupported features listed in Table 1. The use of native methods is also limited. These feature restrictions and native method limitations extend to the entire code base of a SCORE application, including any elements imported from standard Java libraries (e.g., `java.lang`). For example, the Reflection API is not available to the SCORE developer primarily due to its native method dependencies. A somewhat different restriction applies to the method `finalize()` which is available to the Java programmer on a standard JVM. Standard garbage collection is not available on the SCORE. However, a restricted non-preemptive garbage collector can be invoked using a special native method.
3 Embedded System Design

Modern embedded systems can be extremely complex. Their architectures can consist of numerous components cooperating with one another using a variety of communication protocols. In such a framework, system-level functionality can be achieved using several formalisms including: (1) software, (2) microcode, and (3) digital logic.

In its initial stages, a system design can be highly fluid with respect to how formalisms should be used to achieve functionality. As the design progresses, trade-offs must be made between formalisms as well as other design factors. For example,

- What portion of a system-level function should be realized in software?
- What portion of a system-level function should be realized in hardware?
- In what ways should microcode be used to enhance the functionality of hardware?
- What protocols should be employed to implement the various communications that need to take place between components?

Such design decisions can have far-reaching consequences. Complicating the picture is the fact that (1) the individual components making up the architecture are oftentimes developed in parallel by different teams of developers, and (2) component designs are at different levels of completion. The resulting development environment is one where system-level simulation can play a central role facilitating both the specification of the system as well as its overall design.

3.1 Orchestra

Orchestra is a general-purpose system-level modeling environment for discrete-event simulations that is being developed at Sandia National Laboratories. Orchestra embodies the experiences of over a generation of embedded systems development by Sandia engineers.

One of Orchestra’s key strengths is that it provides tremendous visibility into system behavior across a wide range of abstractions. In contrast, standard commercial HDL simulators typically provide visibility into system behavior through a fixed set of view types such as a waveform view and a list view. The waveform view enables system designers to observe relationships between the various signals occurring within an embedded system design as well as signal transitions over time. The list view is a coarse-grained mechanism that can be used to display memory structures and their contents at a given point in time.

In a standard commercial HDL simulator more refined custom views can be developed using scripts (e.g., perl), and interfaces capable of displaying custom views can be developed using GUI toolkits such as Tcl/Tk. For example, a custom view can be created showing changes to the contents of a particular memory over time. The problem one faces when using standard HDL simulators is that the infrastructure for the creation of custom views essentially lies outside of the simulation environment. As a consequence the integration of custom views into the simulation environment is fairly ad hoc and custom views are reduced to second-class citizens of the simulation.

Complexity within an embedded system design amplifies the drawbacks associated with the second-class status of custom views. In a complex system design, it becomes increasingly important to exercise a refined level of control over simulation and to be able to view the simulation along the axis of a particular abstraction. To address these needs, Orchestra provides generalized framework in which predicates can be

---

3 In contrast to a continuous time simulation like Spice.
used to describe properties of system models (i.e., system abstractions). For example, customized breakpoints can be employed to halt a simulation when a simulation satisfies a particular property (e.g., reaches a particular state). In addition to enhanced control, Orchestra promotes custom views to first-class citizenship status facilitating both their specification and display. This enables refined custom views incorporating system context to be created with relative ease. Consider an example where a system contains a FIFO memory that holds communication traffic gathered from an accelerometer. Viewing the data contents of the FIFO alone is a fairly limited view of the state of the system. Including the read and write pointers associated with the FIFO enables an understanding of which data in the FIFO is actually in-scope. Taking it a step further, the in-scope data contents of the FIFO could then be plotted to display the acceleration profile represented by that data. Taking the system context into account results in views at various abstraction levels that are significantly more meaningful and useful than the raw data alone.

Orchestra’s capabilities have enabled Sandia engineers on several occasions to achieve “first-time success in hardware”. By this, we mean the first time the system was put into operation on physical hardware (both FPGA and ASIC), it worked as specified, with no significant problems or bugs. In one case, an Orchestra-based development process led to the discovery of an untenable system design in its early development stages. A traditional development process would have revealed this design flaw only after a hardware user interface prototype had been constructed and exercised, which would have occurred more than a year later, incurring prohibitive cost and schedule impacts.

Within Orchestra a system architecture can be constructed (a process known as elaboration) using a Java-based, special-purpose build language. From a structural standpoint, the resulting (i.e., elaborated) system model consists of a collection of component models connected by communication channels. In this context, a component model is a software artifact capturing the behavior of the component at some level of abstraction (or fidelity). It is important to note that, at the component level, it is possible to develop a number of models for a given component, each having a different level of fidelity. It is also important to note that, at the system-level, a system model can be constructed composed of both low and high fidelity component models. This “mix-and-match” capability enables the overall system-level design to progress in a parallel development environment where individual components may be at different stages of completion.

The ability to model systems and their components across a wide fidelity spectrum distinguishes Orchestra from more narrowly focused simulators such as: instruction set simulators (ISS) and clock-based simulators. For example, Java applications intended for the SCORE platform can be executed within Orchestra using the following four models.

### 3.1.1 Orchestra/JVM

At this level of fidelity, a high-level understanding is developed of how an application program operates within the larger system context. This is done by developing models of other system components with which the target application must interact. These external component models represent system functions, behavior, interfaces, and interconnectivity represented at a level of fidelity required to create an accurate set of interfaces to the target application. This “virtual prototype” system provides an effective framework to explore the communication content and timing requirements associated with data flowing between the target application and other system components. It enables the user to develop large elements of the Java code that will ultimately get deployed into the Score processor. Note that although all system communication timing is simulated during the execution of the system model, the time required to execute the Java application code is not yet represented, and effectively happens instantly in the simulation.
3.1.2 **Orchestra/HL-VSCORE**

The previously developed system models are leveraged at this level of fidelity to complete partitioning the target application into its constituent hardware and software elements. The application software is now simulated using a model of the processor called the high-level virtual score (HLVSCORE). This processor model executes the Java application code from the ROM Image produced by Interlude and now provides a close estimate of the time it takes for the SCORE to execute the Java application code. Additionally, the model also now includes register-accurate models of the I/O modules the processor will be controlling in the system. Simulation proceeds and progress is measured by reaching event boundaries, and at this level those boundaries are at the Java bytecode level. Oscillator clock cycles are still abstracted away and simulation is very fast. It is at this level that migrated Java class libraries are absolutely required and enter into the system design as shown in Figure 2. At this stage final software can be developed and fully tested.

3.1.3 **Orchestra/LL-VSCORE**

In this stage the SCORE processor hardware architecture itself is modeled using the Orchestra Hardware Description Language (OHDL) and the resulting model is called the Low-level Virtual SCORE (LL-VSCORE). This enables oscillator clock cycle timing accuracy and the ability to develop and test the processor’s micro-code (e.g., the micro-code implementation of the byte codes implemented by the SCORE). As a result, the simulation is relatively slow compared to the HLVSCORE or JVM-based simulation and in practice an OHDL simulation will not be significantly faster than an equivalent VHDL or Verilog simulation, but the visibility and controllability available to the user is vastly improved over those tools. The event boundary in this case is at the microcode instruction level, and represents the work done in one clock cycle.
3.1.4 HW-SCORE

This represents the final stage of modeling as a step toward hardware synthesis and is done using the VHDL modeling language and associated toolsets. VHDL will not directly integrate with the Orchestra system model. However, the prior Orchestra modeling stage may be used to automatically generate test vectors that can be read in and used to exercise and verify the VHDL design. Such an approach leverages prior work and assures the level of testing is identical. Essentially, this assures that the previously tested detailed Orchestra model and the final VHDL design are functionally equivalent. Once validated, the VHDL design is synthesized in to digital logic gates and implemented in either a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). In either case, there is a goal to use those same automatically generated testing vectors on a physical tester to verify correct operation of either hardware implementation.

4 Java Source-code Migration

The Java Standard Edition (SE) API provides a number of libraries facilitating the development of Java programs. The three primary library classifications are: (1) Base Libraries, (2) Integration Libraries, and (3) User Interface Libraries. Of these, the Base Libraries are of most interest to our migration efforts. Figure 3 gives a partial listing of the packages found within the Java SE Base Libraries.

Due to its restrictions, the SCORE processor is incompatible with portions of the Java SE Base Libraries. This incompatibility can be resolved through a process that we call migration. The alternative (to migration) is to simply exclude these libraries from the embedded systems software development process entirely; however, such exclusion is undesirable because a strong argument can be made that the abstractions provided by the Java SE Base Libraries are an essential component of high-level Java programming.

Informally stated, the goal of migration is to transform a code base such as a class library into a semantically equivalent form that has the additional property that it is also executable on a targeted platform such as the SCORE processor. Ideally, the migrated and un-migrated versions of a code base would be indistinguishable to the user of the code base. Unfortunately, indistinguishability is a “tall order” and is generally not achievable in practice. As a result, concessions must be made. In particular, users must decide in which cases to accept reduced functionality (e.g., fewer methods) and when to accept altered functionality. Furthermore, these decisions must be made in the context of a larger system design where the properties of a migrated code base must be transparent to the system design and development team. The migration problem can be stated more formally as shown in Figure 4.

There are two types of mechanisms that one can consider in the context of migration: removal
1. Given:
   (a) a set of elements constituting a code base \( C = \{c_1, c_2, \ldots, c_n\} \)
   (b) a set of unsupported elements \( U = \{u_1, u_2, \ldots, u_k\} \)

2. Develop a code base \( C' \) such that:
   (a) \( C' \) maximally preserves the code base of \( C \)
   (b) \( C' \) has no dependencies on \( U \)

**Figure 4:** Statement of the migration problem

and re-implementation. Removal(-based migration) entails strict deletion of code that is not supported by the targeted platform (i.e., the SCORE). From a conceptual standpoint, removal shares similarities with program slicing (Weiser, 1984). Specifically, removal can be seen as the complement of slicing.

On the other hand, re-implementation(-based migration) adapts code by replacing unsupported code fragments with equivalent (or near-equivalent) code fragments expressed in terms of computations supported by the targeted platform. It should be noted that in a practical setting, re-implementation is not always possible.

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

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

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

### 4.1 Removal

Under ideal circumstances, the migration problem can be solved through a strictly removal-based process in which elements such as fields, methods, constructors, and initialization blocks are removed if they contain unwanted dependencies. Though conceptually straightforward, the automated implementation of a removal-based adaptation algorithm is non-trivial in practice. There are three primary reasons for this: (1) the recognition of elements in a code base requires a parser, (2) extensive dependency analysis capabilities
are needed to determine whether an element belonging to the code base depends on an unsupported feature, and (3) a framework needs to be developed automating the actual removal of elements from a code base. It is worth mentioning that it is no small task to create the infrastructure described. However, given such an infrastructure, the removal-based migration algorithm can be completely automated.

Through experimentation, we have determined that a purely removal-based migration algorithm yields unsatisfactory results in practice. The reason for this is that the dependency graph of a code base is typically such that too much gets removed. For example, producing a code base $C'$ that preserves the code base $C$ but that is near empty does not have much practical value. Analysis of the root causes underlying “excessive removal” has revealed that in a variety of cases, a small number of elements are responsible for triggering the removal of a large number of elements in the code base.

Example 2. Every constructor of the class Throwable has a dependency on a native method called fillInStackTrace. If this native method is not supported on the target platform, then the constructors for the class Throwable must either be removed or re-implemented. Removal of the Throwable constructors yields unacceptable results because of its cascading effect on all Exception and Error classes.

We use the term focal point to refer to unsupported elements that trigger the removal of a large number of elements or that trigger the removal of key elements in the code base. Focal points are candidates for re-implementation.

Unfortunately, re-implementation is a manual process and in many cases produces a code base $C'$ whose functionality and/or interfaces are somehow different from the original code base $C$. Thus, the use of re-implementation when solving the migration problem introduces a whole new dimension of issues. Re-implementation involves human judgement. Therefore, a manual process must be developed in order (1) to determine which elements of a code base should be candidates for re-implementation (rather than removal) and (2) to certify the correctness of a re-implemented element.

4.2 Re-implementation

The re-implementation process is outlined in Figure 5. It has several key activities: impact analysis, focal point identification, evaluation, code analysis and redesign of focal points, and transformation.

4.2.1 Impact Analysis

In the impact analysis phase, the affected elements are identified. Affected elements include classes that use unsupported elements as well as all program elements that directly or indirectly depend on these classes. A necessary first step before this identification process is type resolution which disambiguates identical names. This produces an equivalent code base with canonical names for classes and interfaces and fully qualified names for all other identifiers. Each attribute and method is then analyzed to identify which ones use unsupported elements. The transitive closure is then computed based on the usage relations to these affected elements.

4.2.2 Focal Point Identification

After affected classes have been identified, focal point analysis identifies the most likely focal points. There are several strategies for identifying these candidates:

1. Rank the affected elements based on the number of program elements that transitively depend on
them. From this, the set that has the largest number of transitive dependencies is selected.

2. Assess the relative contribution of each element to the overall dependency graph. One way to do this is to model the dependency graph as a Markov chain and calculate the steady state distributions of the transitions. The elements can be ranked based on their contribution. This approach has been used in component ranking (Inoue et al., 2005).

3. Identify the elements with relatively high dependencies by manipulating the dependency as a design structure matrix (DSM) (Steward, 1981). A partitioning algorithm can be used to divide up the matrix into groups of tightly related elements, with groups that have few dependencies percolating to the top of the matrix and those that are most dependent drifting to the bottom of the matrix. The overall degree of dependence on any particular element can be assessed by its relative position in the matrix. Partitioning can be achieved through an efficient triangularization algorithm (Kusiak et al., 1994).

4.2.3 Evaluation

Evaluation is a manual activity where the various focal point measures calculated in the previous steps are studied, along with graph visualizations of the impacted elements vis-a-vis the entire dependency graph. In addition, specific requests to preserve certain APIs are also accounted for. The outcome of this analysis is the disposition of the candidate focal points, that is, whether they can be removed or need to be re-implemented. After this activity, a report is developed explaining the rationale for each decision to re-implement. The rationale provides an audit trail in case such a decision needs to be revisited in the future.

4.2.4 Code Analysis and Redesign of Focal Points

Code analysis is a tool-supported manual inspection process whose goal is to provide designers with an understanding of the nature of dependencies on unsupported elements. Such dependencies will have to be removed on a case by case basis in a manner that minimizes the impact on the way those focal point elements are to be used. From previous migration efforts (Winter & Mametjanov, 2007), 3 high level strategies have been identified:

1. Deletion of affected expression. If the affected expression has little or no effect on subsequent data flow, it is feasible to simply delete it. In general, this would be useful for statements that only display output.

2. Replacement of affected expression by equivalent or near equivalent expressions and statements. For example, since native methods are not supported, the functionality can be replaced by equivalent statements written in Java. If the affected expression has significant effect on subsequent data flow, it cannot be simply deleted but needs to be rewritten without the affected expression. If it cannot be shown that the rewrite preserves the behavior, then extensive testing of dependent classes and methods will need to be performed.

3. Conversion from float to int types. As floating point is unsupported, it may be possible to replace them with integer types. However, if the truncation is unacceptable, a fraction representation with a numerator and denominator may be considered. Unlike other replacements, this requires dataflow analysis to identify and appropriately modify all affected uses.

Specific instances of these replacements are then manually encoded into transformation rules. In addition, rationales for each rule are embedded as JavaDoc-style comments to be generated into the migrated code for future reference.
It is worth mentioning that the primary goals of redesign are (1) dependency removal, and (2) correctness preservation. The intent is to achieve re-implementation through highly localized changes. This facilitates the assurance argument, because correctness of the re-implementation follows from the correctness of the localized changes. However, this approach may give rise to situations where re-implemented code is less efficient than the original. In extreme cases, the inefficiency introduced during re-implementation may be such that the real-time constraints of the application are threatened. Under such circumstances, a more global kind of re-implementation may need to be considered (e.g., a complete redesign of a method).

4.2.5 Transformation

The transformation step takes the rules written in the previous step and applies them to the code base to be migrated.

4.2.6 Evolution Issues

Having a set of transformation rules, migration of subsequent releases of the library would be less effort intensive. To simplify matters, the automated steps of re-implementation will be re-run to produce a new set of affected elements and focal point candidates. As libraries tend to preserve the functionality, it is expected that the focal point candidates would not change greatly. Only the new focal points needs to be manually examined and a rationale provided for their disposition. Code analysis and redesign need only be carried out over those new focal point elements determined to require re-implementation and new transformation rules are written for them. A potential complication is when certain code changes in the new release may invalidate some existing transformation rules. Such rules will need to be readjusted manually.

5 Monarch

Our approach to code migration is transformation-based and operates at the source code level. In particular, we are developing a tool called Monarch that has the following properties:

- Removal is fully automatic.
- Re-implementation is manual, but a re-implemented component is then encoded as a replacement transformation and hardwired into the migration. This enables automatic end-to-end replay of a migration. This replay capability can be useful for migrating new library releases.
- Migration produces pretty-printed Java source-code conforming closely to Java coding conventions (Sun Microsystems, 1997).
- Migration has the ability to preserve as well as add JavaDoc comments. For example, JavaDoc comments can serve as the vehicle for introducing trace information into the migrated source code indicating which elements were removed and why. Such information can facilitate assurance-based activities such as code review.

Monarch migration is shown in Figure 5 and consists of two phases: (1) a replacement phase in which essential components of a code-base are replaced by suitably re-implemented components, and (2) a removal phase in which components having undesired dependencies are removed. Section 5.3 describes the

\[^{4}\text{Sun typically releases a new version of its libraries every 18 months.}\]
removal phase of Monarch migration. The replacement phase is elided because hardwiring re-implemented components as transformations is relatively straightforward. The difficulty in replacement lies in the re-implementation of components (discussed in Section 4.2) not in their encoding.

5.1 The Bascinet/TL System

For over a decade, we have been developing a framework called the TL System (Winter, 2007) (Winter & Subramaniam, 2004), a framework for expressing transformation-based computation. The TL system includes (1) a GLR parser generator, (2) a TL language interpreter, (3) an SML compiler, and (4) an abstract pretty-printer. The TL System provides an environment in which computation can be described in a hybrid fashion – as a mixture of (higher-order) transformations written in a language we have developed called TL, and functions written in SML. Such a hybrid program is called a TL program and the data it primarily operates on are parse trees.

Aside 2. Transformation is the term used when referring to the computation articulated by a TL program. In a more general setting, a system supporting the articulation of transformations is called a transformation system. In contrast to imperative systems, where computation is assignment-based, computation in a transformation system is based on rewrite rules. It should be noted that transformation systems differentiate themselves from (pure) rewrite systems by providing explicit primitives for controlling the application of rewrite rules to trees (which are also known as terms).

The development of TL programs is supported by an in-house developed IDE called Bascinet written
in Java\textsuperscript{5}. From a conceptual standpoint, “Bascinet is to TL” as “Eclipse is to Java”. Bascinet supports a number of activities that are essential for scaling transformation-based programming to real-world problems. In addition to integrating third-party software such as text editors, browsers, and graphical display tools, Bascinet provides system-level support for applying TL functions to file hierarchies. There are three TL System functions that can be applied to a file hierarchy: (1) parse the contents of a file hierarchy, (2) transform the contents of a file hierarchy, and (3) pretty-print the contents of a file hierarchy. To facilitate application of TL System functions to file hierarchies, Bascinet provides the ability to define include/exclude sets thereby controlling the application of transformations to files having specific extensions, and provides meaningful feedback about application failures occurring within a file hierarchy.

The notion of transforming a file hierarchy can be further refined into two application modes: discrete application and continuous application. In a discrete application mode, the files in the file hierarchy are transformed independently - one transformation is applied to one file and then the application process is repeated. In a continuous application mode, the files in the file hierarchy are transformed collectively - a single (stateful) transformation spans all selected files. Continuous application is well-suited when transformation goals span multiple files (e.g., dependency analysis within a Java code base).

Both the TL System and Bascinet are freely available on-line (The TL System, 2010).

5.2 Java Infrastructure

Based on the TL System, we have developed a Java infrastructure consisting of the following:

- A Java parser that preserves JavaDoc comments.
- A Java pretty-printer that formats Java parse trees in a manner conforming closely to Java code conventions (Sun Microsystems, 1997).
- A framework where it is easy to write transformations that gather a variety of metrics over a code base (e.g., number of classes, number of interfaces, lines-of-code, number of field declarations, etc.).
- A transformation that constructs an internal model of a Java code base forming the foundation of dependency analysis.
- A general ability to produce dot-files showing interesting information. These files can be viewed via the ZGR viewer (Pietriga, 2005), which has been integrated into Bascinet. Currently, we have a transformation that when applied to a file hierarchy produces a dot-file showing the field declarations within classes as well as their inheritance and interface relationships. We also have transformations that gather a variety of metrics including those shown in Table 2.

5.3 Monarch’s Removal Phase

In this section we sketch our approach to the removal phase of Monarch migration. It is important to mention that the removal phase assumes that re-implementation based replacement has already been carried out. Specifically, all remaining unsupported dependencies should now be (strictly) removed. The discussion that follows captures the essence of what occurs during Monarch’s removal phase, but is incomplete and a number of technical details are abstracted away.

We begin with a migration model in which there are three term structures of interest:

\textsuperscript{5}In a prior implementation, the Bascinet/TL System composition was referred to as the HATS System.
### Java Library

<table>
<thead>
<tr>
<th></th>
<th>java.io</th>
<th>java.lang...</th>
<th>java.math</th>
<th>java.nio...</th>
<th>java.util...</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>23</td>
</tr>
<tr>
<td>Compilation Units (CUs)</td>
<td>84</td>
<td>169</td>
<td>7</td>
<td>139</td>
<td>232</td>
<td>624</td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>29003</td>
<td>54890</td>
<td>9422</td>
<td>46862</td>
<td>116986</td>
<td>257163</td>
</tr>
</tbody>
</table>

### Type Uses in Field Declarations (classes, enums and interfaces)

<table>
<thead>
<tr>
<th></th>
<th>type uses consisting of simple ids</th>
<th>total type uses</th>
<th>Percentage of uses that are simple ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>type uses consisting of simple ids</td>
<td>392</td>
<td>395</td>
<td>99.2%</td>
</tr>
<tr>
<td>total type uses</td>
<td>676</td>
<td>685</td>
<td>98.7%</td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>85</td>
<td>98.8%</td>
</tr>
<tr>
<td></td>
<td>215</td>
<td>215</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>1802</td>
<td>1847</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>98.2%</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enums</td>
<td>0</td>
<td>4</td>
<td>99.2%</td>
</tr>
<tr>
<td>Interfaces</td>
<td>12</td>
<td>29</td>
<td>99.5%</td>
</tr>
</tbody>
</table>

### Classes

<table>
<thead>
<tr>
<th></th>
<th>inner classes (non-static)</th>
<th>static nested classes</th>
<th>classes within methods</th>
<th>classes within constructors</th>
<th>top-level classes</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>type uses consisting of simple ids</td>
<td>4</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>73</td>
<td>107</td>
</tr>
<tr>
<td>implicit extensions to Object</td>
<td>0</td>
<td>28</td>
<td>1</td>
<td>0</td>
<td>125</td>
<td>258</td>
</tr>
<tr>
<td>type uses consisting of qualified ids</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>100</td>
<td>132</td>
<td>6</td>
<td>0</td>
<td>132</td>
<td>492</td>
</tr>
</tbody>
</table>

### Type Uses in Class Extensions

<table>
<thead>
<tr>
<th></th>
<th>type uses consisting of simple ids</th>
<th>implicit extensions to Object</th>
<th>type uses consisting of qualified ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>type uses consisting of simple ids</td>
<td>65</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>implicit extensions to Object</td>
<td>87</td>
<td>72</td>
<td>2</td>
</tr>
<tr>
<td>type uses consisting of qualified ids</td>
<td>110</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>total</td>
<td>305</td>
<td>180</td>
<td>7</td>
</tr>
</tbody>
</table>

### Percentage of uses that are simple ids or Object

|                              | 99%                              | 98.7%                         | 100%                                    | 97.7%                   | 98.6%             | 98.5%             |

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enums</td>
<td>0</td>
<td>4</td>
<td>99%</td>
<td>98.7%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td>12</td>
<td>29</td>
<td>100%</td>
<td>97.7%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Syntax-based metrics for selected Java core API classes

1. **Unsupported fragments** - This set describes, at the syntactic level, the elements of Java that are not supported by the target platform (e.g., keyword `float`).

2. **Composite elements** - The elements of this set are types. That is, *classes, enums, and interfaces*. This includes composite elements that are nested (i.e., static and inner) within other composite elements. Composite elements form contexts within which migration takes place.

3. **Atomic elements** - The elements of this set are *fields, methods, constructors, and initialization blocks*.

A partial grammar defining the syntactic categories of our abstracted Java elements is given in Figure 6. Migration treats atomic and composite elements differently. In particular, based on the results of dependency analysis, migration of an *atomic element* produces one of the following outcomes.

1. **Remove** - the atomic element is removed completely. It should be noted that (complete) removal represents a conservative approximation by our migration algorithm. This approximation can be
composite ::= element_list

element_list ::= element element_list | ε

element ::= atom | composite

atom ::= field-declaration | method-declaration | constructor-declaration | instance-initializer | static-initializer

Figure 6: Syntactic Categories

compensated for by re-implementation.

2. *Copy* - an atomic element having no unsupported dependencies is migrated fully intact (i.e., untouched).

The migration of a *composite element* is defined in terms of the migration of the elements of which it is comprised. The structural skeleton of a composite element is always *copied* during migration (i.e., the notion of a composite element is supported by the SCORE). This skeleton is then populated with the migrated contents of the original composite element.

Within a model, every element is associated with a key. Composite elements are associated with their *canonical name* (Gosling et al., 2005). Fields are associated with their *fully qualified name*. Methods and constructors are associated with their *fully qualified names* together with their signatures. And initialization blocks are associated with *internal keys* generated by Monarch.

We use the term *resolution* to describe the translation of an element into its key. Resolution is a fundamental operation guiding Monarch migration. The analysis capabilities required to perform resolution are non-trivial and encompass scope rules, access control rules, import rules, type resolution, and inheritance. Figure 7 shows a piece of code highlighting some of the complexity associated with resolution. For example, in this figure the key `p1.A.A1.z1` is associated with the field `z1` which is declared in the inner class `A1`. It is important to note that the central difficulty in resolution revolves around locating the *definition* corresponding to the *use* of an element (e.g., find the declaration of class `C2` being referred to in the initialization of `z2`). Summarized from a conceptual standpoint, analysis is oriented around *use* of an atomic element while removal is oriented around its *definition* (*def*).

Within Monarch, resolution is performed by a function named `resolve` that is implemented in SML. The migration of atomic elements (i.e., their removal or copy) is then accomplished via transformations whose application is controlled by the `resolve` function.

Analysis of atoms must account for direct as well as indirect dependencies on unsupported language features. In Monarch, this is modeled by a set of keys $R_{C}$ representing the transitive closure of atomic elements that depend on unsupported features of the SCORE. The set $R_{C}$ is the smallest set that is closed under the axioms and rule given in Figure 8. In these axioms, the terms `FpLiteral`, `strictfp`, `float`, and `double` denote terms corresponding to Java token-level floating point dependencies (which is the only unsupported feature assumed in this discussion).

The T-transitive rule shown in Figure 8 uses an ellipsis-based notation to denote nontrivial aspects related to the recognition of term structure. Specifically, the term `atom[...use...]` denotes a parse tree corresponding to an atomic element containing *use* as a sub-tree. The ellipsis notation glosses over a number
of things including the scope boundaries as well as shadowing that may exist between atom, the root of the term, and the sub-term use. Furthermore, the variable use captures the entire class of references to fields, methods, and constructors. In practice, the treatment of fields and methods/constructor require separate implementations. And finally, the notation used in the rule also abstracts away contextual information needed by resolve. The actual removal transformation we have developed contains portions that explicitly keep track of this information.

\[
\begin{align*}
FpLiteral & \in R_C & \text{(Axiom-lit)} \\
\text{strictfp} & \in R_C & \text{(Axiom-strictfp)} \\
\text{float} & \in R_C & \text{(Axiom-float)} \\
\text{double} & \in R_C & \text{(Axiom-double)} \\
\text{atom[...use...]} & \text{resolve(use)} \in R_C & \text{(T-transitive)} \\
\text{resolve(atom)} & \in R_C
\end{align*}
\]

Figure 8: Dependency rules for floating point
Using the dependency set $R_C$, the removal of atoms from a class can be concisely stated by the rule T-remove shown in Figure 9. As was previously mentioned, removal is oriented around atom definitions ($def$). Informally stated, the rule says that definitions of atoms containing dependencies on unsupported features are to be removed.

$$\text{resolve}(def) \in R_C \Rightarrow \text{composite}[\ldots def \ldots] \rightarrow \text{composite}[\ldots \varepsilon \ldots]$$

(T-remove)

Figure 9: Removal

Our current approach to migration is an extension of our previous migration efforts. Previously (Winter et al., 2007) (Winter & Mametjanov, 2007), a transformation-based approach to migration was explored in which dependency analysis was light-weight and performed exclusively via transformations. At that time, the HATS transformation system (Winter & Beranek, 2006) was used to perform migration. The light-weight analysis used was incomplete and as a result the removal phase of migration required additional manual effort.

Our current approach incorporates lessons learned from our previous work. This has led to a significant re-design of HATS resulting in the Bascinet/TL System. In the context of this discussion, there are two changes worth mentioning: (1) our migration system now contains an SML component that performs a full (i.e., heavy-weight) dependency analysis of the source code, and (2) Bascinet’s continuous application mode, described in Section 5.1, is seen as a key enabler of analysis and migration of code bases that are distributed across a large number of files.

6 Related Work

Rayside and Kontogiannis (Rayside & Kontogiannis, 2002) 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 the class loader for the SCORE (Morrison, 2005) already 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.

Similar optimizations have also been explored by Tip et. al. (Tip et al., 2002) with the goal of reducing application size for distributed as well as embedded environments. The compression techniques used in (Tip et al., 2002) include method inlining, class hierarchy transformation, removal of unreachable methods and removal of redundant fields. On average, this resulted in applications being compressed to 37.5% of their original size.

6The Bascinet/TL System is a descendent of the HATS system.
Unsupported Keywords

<table>
<thead>
<tr>
<th>native</th>
<th>synchronized</th>
<th>transient</th>
<th>volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>strictfp</td>
<td>enum</td>
<td>assert</td>
<td></td>
</tr>
</tbody>
</table>

Unsupported Types

| char     | double       | float     | long     |

Table 3: List of items not supported by the Java Card

6.1 VMs

Java Card v2.2.2: The Java Card platform is a subset of the JVM targeting resource-constrained devices. Programs for the Java Card platform must be written in a subset of Java informally referred to as the Java Card platform language. For the most part, this language can be characterized at the syntactic-level as a subset of the Java syntax and Java Card applications (called Java Card applets) can be compiled using a standard Java compiler. However, in order to execute on a Java Card platform, the class files comprising a Java Card applet must be converted during a post-processing stage. This conversion combines all the class files into a single file and translates bytecodes into a compressed format. The resulting file is called a CAP file.

The assumptions under which the Java Card is expected to operate prohibit support for string manipulation, thread management and floating point arithmetic. A number of other features are also not supported such as: (1) dynamic class loading, (2) threads, (3) enumerated types, (4) enhanced for-loops, (5) assertions, and (6) reflection. On the semantic level, a noteworthy characteristic of the Java Card Platform is that it also places a number of (semantic) restrictions on access control in Java packages (e.g., the circumstances under which modifiers such as public, protected and “package-private ” may be used).

A list of the unsupported keywords and types is given in Table 3. In addition, the Java Card Platform supports only a few classes from the java.lang package. All other Java programming language core API classes are not supported.

Squawk: This is a CLDC (Sun Microsystems, 2003) compliant implementation of the JVM targeting “very small devices” which the authors define as: “32-bit processors having no more than 8 KB of RAM, 32 KB of non-volatile memory, and 160KB of ROM”. In order to be CLDC compliant, a VM must support a variety of functions such as garbage collection, dynamic class loading, as well as verification. The Squawk VM (Shaylor, 2003) is implemented primarily in Java and the major design challenges it faced was how to achieve CLDC compliance given the limited resources available to very small devices. Much of Squawk design focuses on (1) reducing classfiles in a preprocessing stage, (2) encoding bytecodes in a more compact form, (3) transforming code to simplify garbage collection, and (4) eagerly performing symbolic resolution.

7 Conclusion

Modern embedded systems are developed within a fluid design space in which system functionality can be achieved using a range of formalisms including (1) high-level software, (2) microcode, and (3) customized hardware. In this setting, broad-spectrum simulation plays an essential role enabling exploration of the design space.
In order to be cost-effective, an embedded system design process must also leverage general-purpose tools such as IDEs and unit testers to the fullest extent possible. In this regard, a primary goal of broad-spectrum simulation is to align the desktop development environment with the development environment of the (physical) embedded system.

Another important dimension of “desktop-physical system” alignment can be provided by the Java SE Basic Libraries. These libraries should be made available to the embedded systems programmer to the fullest extent possible. To support this migration process, we are developing tools that judiciously remove or reimplement portions of the Java library that are not supported by the target embedded processor. Just as importantly, restrictions on the libraries should be propagated back to the early stages of the design. These two objectives are met by Monarch and Orchestra.

References


