

Transformation-based Library Adaptation for Embedded Systems*

Victor L. Winter Azamat Mametjanov
University of Nebraska at Omaha
{vwinter,amametjanov}@mail.unomaha.edu

Steven E. Morrison James A. McCoy Gregory L. Wickstrom
Sandia National Laboratories
{semorri,jamccoy,glwicks}@sandia.gov

Abstract

Embedded systems are computational environments having restricted capabilities. These restrictions make the incorporation of high-level general purpose libraries, such as `java.lang` and `java.util`, into the embedded systems software development process problematic. This paper describes a general transformation-based approach that can be used to adapt Java libraries making them compatible with the computational restrictions imposed by embedded environments.

1 Introduction

Embedded systems span a broad spectrum of applications ranging from medical applications, weapons systems, space borne systems to flight control systems. The increasing complexity of these applications has created an attendant demand for high-level language support for embedded systems programming. This paper takes a look at one aspect of the high-level language support problem: the adaptation of general purpose libraries with the goal of making them suitable for use in embedded applications. In particular, a transformation-based technique is described capable of automatically carrying out the majority of the activities needed to adapt Java libraries such as `java.lang` for use with an embedded system known as the SCORE processor [11, 18], a hardware implementation of a significant subset of the Java Virtual Machine that has been developed at Sandia National Laboratories.

Today, the computational underpinnings of a variety of embedded systems are based on the Java Virtual Machine

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

(JVM). In essence, what this means is that computing capabilities, at the hardware level, are provided via the execution of Java bytecodes. In turn, these Java bytecodes are given a semantics with respect to the JVM. The result is an embedded system that can either (1) execute Java class files directly, or (2) execute a slightly modified version of Java class files. A classic example of such an embedded computing platform is the Java Card technology [12] where a restricted implementation of the JVM is placed on a smart card, which can then directly execute Java class files. Incidentally, the Java Card platform supports an extremely limited set of Java libraries.

The hardware in embedded systems is oftentimes subjected to heavy constraints. In addition to economic forces, physical limitations place strict bounds on various hardware attributes such as volume, weight, and power usage. As a result, the computational capabilities of embedded platforms generally represent a scaled-back version of the more general purpose computing platforms found on a PC. This holds true for embedded systems whose computational capabilities are based on the JVM. Computational restrictions on the JVM are made precise at the bytecode level. In general, a computational platform X represents a restricted version of the JVM if the set of bytecodes supported by X is a strict subset of the set of bytecodes supported by the JVM. In this paper, we will use the term `jVM` to refer to a virtual machine representing an instance of the JVM that is restricted in the manner just described.

Listed below are examples of some of the restrictions that are typically placed on embedded implementations of the JVM:

1. **prohibited use of strings** – The direct storage of strings is a very memory intensive operation. This is compounded by the fact that many `jVM`'s do not support garbage collection.
2. **prohibited use of floating point arithmetic** – The im-

plementation of floating point arithmetic in hardware is a well-known source of errors. Thus, the presence of hardware implementing floating point arithmetic introduces a serious assurance burden for the embedded system.

3. **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.
4. **prohibited/restricted ability to execute native methods** – Native methods are used by the libraries whenever a required operation cannot be programmed in Java. Typically, such operations are platform-dependent, e.g. hashing of a memory location. Different embedded JVMs may support different sets of native methods.

In general, the restrictions associated with a given JVM determine the subset of Java which the JVM is capable of executing. For example, if a given JVM does not support floating point operations, then Java code fragments (e.g., in fields, methods, constructors, etc.) containing such operations are inadmissible.

The computational restrictions imposed by a JVM poses a problem with respect to goal of providing embedded systems programmers access to general purpose libraries such as `java.lang` and `java.util`. The option of excluding general purpose libraries from the embedded systems software development process entirely is undesirable because a strong argument can be made that the abstractions provided by the Java libraries are an essential component of high-level Java programming. Therefore, it is highly desirable to incorporate the use of libraries in the development of embedded applications to the extent possible. However, it is also the case that substantial libraries such as `java.lang` and `java.util` typically make full use (or near full use) of the capabilities of the JVM and thus – in unaltered form – are unsuitable for execution on a given (restricted) JVM. In other words, an incompatibility exists between the JVM and a Java library making full use of the JVM's computational capabilities. This incompatibility can be resolved through a process that we call *library adaptation*.

There are two types of approaches that one can consider in the context of library adaptation: *removal* and *re-implementation*. Removal entails strict deletion of code that is not supported by a JVM. Re-implementation adapts the libraries by replacing (when possible) unsupported code

fragments with equivalent code fragments that are expressed in terms of computations supported by the JVM. It should be noted that re-implementation is not always possible. Furthermore, re-implementation of libraries is a potentially error-prone approach because it involves the development of new code. Depending on the nature of the re-implementation (i.e., its scope and complexity), re-implementation can entail a potentially unacceptable risk of introducing bugs into the resulting adapted library. Standard libraries typically undergo a significant number of tests to ensure their correctness before official release by Sun. Re-implementation would jeopardize assurance arguments based on these tests. In addition, standard libraries mature based on bug feedback from a large group of library users. Re-implemented libraries used in embedded applications generally will not be subjected to the maturation process associated with a large user base.

Because of the concerns raised in the previous paragraph, library adaptation based on the *removal* (of unsupported code) is more attractive than adaptation based on *re-implementation*. At least from the perspective of assurance. Abstractly, *removal* can be understood simply as a process of deleting code fragments from a library containing computations that are unsupported by the given JVM. Such a perspective reveals that removal shares similarities with program slicing. More specifically, removal can be seen as the complement of slicing. In a more theoretical setting, removal can be described as a relation between unsupported computational elements and code fragments. It is the calculation of the transitive closure of this relation that gives rise to the complexity of removal-based library adaptation.

Library adaptation based on removal can be done manually. However, manual removal-driven library adaptation is a labor-intensive task. As an experiment, we have manually migrated one Java library – the system package `java.lang` (not including the sub-packages) and it took about one man-month of work. As of this writing, the standard libraries consists of 203 public packages. The impracticality of manual adaptation is further emphasized by evolution of the libraries. Major updates to the libraries (e.g. v. 1.5 to 1.6) are released approximately once a year. Since it is desirable to provide embedded systems programmers with up-to-date versions of the libraries, a newer release warrants a new iteration of library adaptation. This scenario makes a strong argument for the automation of library adaptation.

Contribution In this paper, a transformation-based approach for the adaptation of Java libraries is presented. This approach is currently being developed to create a tool that is able to adapt Java libraries (e.g. `java.lang` and `java.util`). The targeted platform for adapted libraries

is the SCORE processor – a JVM developed at Sandia National Laboratories [11, 18].

The approach developed in this paper is not restricted to a particular embedded platform or to the libraries. First, other embedded platforms might support a different instruction set. As long as an unsupported feature can be tied to a particular source code construct, we can remove such unsupported code: e.g., the floating-point arithmetic feature is tied to float/double source code declarations. Second, the targeted platform can evolve to support a larger instruction set. In such cases, we can re-run the adaptation so that the newly supported feature is not excluded from the original libraries. Finally, the libraries represent collections of Java code. Though applied to libraries, our approach is not restricted to libraries; any legal Java code can be adapted.

The remainder of this paper is organized as follows: Section 2 describes adaptation in a semi-formal setting. Section 3 discusses the infrastructure needed to realize adaptation in practice. Section 4 describes the results of adapting the library `java.util`. Section 5 discusses related and future work, and section 6 concludes.

2 Adaptation

The adaptation of Java libraries can be seen as a transformation-based process that takes as input a set of classes (e.g., a library) L_1 compatible with a standard JVM, and produces as output a set of classes (e.g., an adapted library) L_2 compatible with a restricted implementation of the JVM. In general, a set of classes L is compatible with a given JVM if the bytecodes in the compiled version of L (i.e., its classfiles) belong to the set of bytecodes implemented by the JVM.

2.1 Constraints and Considerations

The adaptation problem can be described in a more formal setting as removing *terms* (i.e., components) from a term structure C corresponding to a class. In this setting, the removal of terms is subject to certain syntactic and semantic constraints that will be discussed shortly. We write $t[\dots t_i \dots]$ to denote that the term t contains t_i as a sub-term. Even though the notation $t[\dots t_i \dots]$ contains ellipsis and is only informally defined, we believe that in the context of a transformation-based discussion it is conceptually clearer than either writing $t_i \in t$ or using the positional notation that is often used in a more traditional term rewriting setting. Using the notation $t[\dots t_i \dots]$, a removal-based adaptation of a class t belonging to a set of classes L can now be expressed as the selective application of *transformation steps*¹ having the form:

$$\mathcal{T}_i : t[\dots t_i \dots] \rightarrow t[\dots \epsilon \dots] \quad (1)$$

For example, suppose the JVM we are targeting does not support the native method `Object.hashCode()`. Then, the transformation that removes the declaration of this method can be expressed abstractly as an instance of the transformation \mathcal{T}_i in which t is instantiated to the class `Object` and t_i is instantiated to the declaration of the method `hashCode`. This would yield an abstract transformation having the following form:

$$\begin{aligned} & \text{Object}[\dots \text{public native int hashCode();} \dots] \\ & \rightarrow \\ & \text{Object}[\dots \epsilon \dots] \end{aligned}$$

In this framework, control over the application of transformation steps like \mathcal{T}_i is subject to four primary constraints:

1. Each transformation step is required to produce a term that is syntactically well-formed;
2. When applied to a term $C_{initial}$ denoting a class the transitive closure of transformation steps must yield a resulting term C_{final} such that $C_{final} \sqsubseteq C_{initial}$, where \sqsubseteq denotes refinement with respect to a given semantic function \mathcal{M} (e.g., the JVM);
3. After transformation, all terms must consist solely of elements that are supported by the targeted computing platform;
4. The removal of terms t_i must be *minimal* with respect to certain assumptions (to be discussed shortly).

The syntactic well-formedness constraint suggests that only terms having particular shapes be considered as candidates for removal. For example, if the objective is to remove dependencies on floating point values and operations then simply removing all terms denoting floating point values will yield results that are not syntactically well-formed. In Figure 1, a number of examples are shown in which such an approach towards removal leads to syntax errors. These examples suggest that in order to assure syntactic correctness removals should be performed with respect to larger term structures such as assignment statements, field declarations and method declarations.

While limiting removal to such terms yields results that are syntactically well-formed, the goal of producing a semantic refinement places additional constraints on which terms should be removed. In other words, syntactic well-formedness alone does not form a sufficient criterion for the basis of removal. In the second column of Figure 1, two examples are shown in which removals produce results that

¹In general, a *transformation step* distinguishes itself from a *rewrite step* because it may be realized, in a nontrivial fashion, in terms of a num-

ber of rewrite rules and strategies.

Removals Leading to Syntax Errors	Removals Leading to Semantic Errors
<code>int x = (int) 1.0 ;</code> → <code>int x = (int) ;</code>	<code>int x = 0;</code> <code>int f(){ double x = 1.0; return (int) x;}</code> → <code>int x = 0;</code> <code>int f(){ return (int) x;}</code>
<code>double x;</code> → <code> x;</code>	<code>int ask() {return (int)answer();}</code> <code>double answer(){return 3.33;}</code> → <code>int ask() {return (int)answer();}</code> <code> </code>
<code>float foo() {...}</code> → <code>foo() {...}</code>	
<code>if (y < 1.0) x = 0;</code> → <code>if (y <) x = 0;</code>	

Figure 1. Removal units leading to syntactic/semantic errors

are syntactically correct but which do not satisfy the semantic well-formedness constraint defined by the relation \sqsubseteq .

And finally, the minimality constraint requires that a good-faith attempt be made to retain as much of the original code base as possible. Otherwise, an empty code base could be used to trivially satisfy the constraints 1, 2, and 3. We address the minimality constraint by removing code at the level of class members (fields, methods, etc.). Based on the inspection of several Java libraries, we have found that code within class members is tightly-coupled, while class members within a class are loosely-coupled. For example, removal of a statement within a member will typically invalidate the entire member, while removal of a member within a class will not necessarily invalidate the entire class.

2.2 A Deductive Description of Removal-based Adaptation

The deductive description presented here is based on a conceptual model in which there are four term structures of primary interest: unsupported features (U), classes (C), members (M), and keys (K). Terms corresponding to unsupported features U describe (at the syntactic level) the primary set of unsupported features of the target JVM. Terms corresponding to classes C are list-like structures whose elements are members M . Based on their structural properties, we partition member terms into two kinds: (1) atomic members and (2) composite members. The set of *atomic members* consists of field declarations, method declarations, constructor declarations, as well as class blocks (i.e., instance initializers and static initializers). The set of *composite members* consists of inner class declarations. An abstract syntax defining the syntactic categories of members is shown in Figure 2.

M	::=	$x \mid mb$
x	::=	inner-classes
mb	::=	$m \mid b$
m	::=	field-declaration method-declaration constructor-declaration
b	::=	instance-initializer static-initializer

Figure 2. Syntactic Categories for Class Members

Our adaptation algorithm treats atomic and composite members differently. In particular, based on the results of dependency analysis, an atomic member is either removed completely or remains fully intact (e.g., a field/method/constructor declaration is either in or out). The decision to subject atomic members to such a removal policy represents an approximation by our removal algorithm.

In contrast, adaptation selectively removes portions (i.e., atomic (sub)members) from classes as well as composite members occurring as members of classes.

With the exception of class blocks, all atomic member terms are associated with a key K by which they can be referenced. In the case of fields, keys denote the fully qualified field identifier. In the case of methods (and constructors), keys denote the fully qualified method (constructor) identifier and their signature.

```

class A {
  double x = 1.0;
  int y;
  int z;
  { int x = (int)this.x; y = x; }
  { z = (int) x; }
}

```

Figure 3. References to the member `double x = 1.0`

A reference to a member can only occur under certain environmental conditions (i.e., from within certain scopes). We write $ref\text{-}to(m)$ to denote a function that, when given an atomic member m , returns the key by which m can be referenced. The following is a concrete example showing the relationship between a member m and its key.

```

m = public native int hashCode();
ref-to(m) = Object.hashCode()

```

Let $t = ref\text{-}to(m)$. We write $x[\dots\hat{t}\dots]$ to denote an occurrence of t in an environmental context where t is interpreted as a reference to m . In Figure 3, an example is given showing a code fragment containing several occurrences of `x`. Occurrences that constitute references to the atomic member m where m is `double x = 1.0` are highlighted.

Our goal is to develop a transformation that removes a member term t from a class C when t directly (or indirectly) depends upon a feature that is unsupported on the target computing platform. It is important to note that this removal of members from classes is fully extended to inner classes, but does not extend to inner classes nested within inner classes. Empirical evidence suggests that nested inner classes are rare. In such corner cases, a comment is logged and treatment of the particular case is left to manual adaptation.

Removal of terms referencing unsupported features needs to account for transitive dependencies. To this end, we define a term set R_C that contains the closure of transitively dependent member terms. Membership in R_C is defined by the set of axioms and rules, presented in a natural deduction-style syntax, shown in Figure 4. In the axioms, the terms $FpLiteral$, $Modifier[\text{strictfp}]$, $BasicType[\text{float}]$, and $BasicType[\text{double}]$ denote terms corresponding to Java token-level floating point dependencies. The set R_C is the smallest set that is closed under the rules given in Figure 4.

Using the dependency set R_C , the removal of members

$$\frac{}{FpLiteral \in R_C} \text{ (Axiom-lit)}$$

$$\frac{}{Modifier[\text{strictfp}] \in R_C} \text{ (Axiom-strictfp)}$$

$$\frac{}{BasicType[\text{float}] \in R_C} \text{ (Axiom-float)}$$

$$\frac{}{BasicType[\text{double}] \in R_C} \text{ (Axiom-double)}$$

$$\frac{C[\dots m \dots] \quad t \in R_C \quad m[\dots \hat{t} \dots]}{ref\text{-}to(m) \in R_C} \text{ (T-transitive)}$$

Figure 4. Dependency rules for floating point bytecodes

from a class can be concisely stated by the rules T-adapt1 and T-adapt2.

$$\frac{C[\dots mb \dots] \quad t \in R_C \quad mb[\dots \hat{t} \dots]}{C[\dots mb \dots] \rightarrow C[\dots \epsilon \dots]} \text{ (T-adapt1)}$$

$$\frac{C[\dots x \dots] \quad t \in R_C \quad x[\dots \hat{t} \dots]}{x[\dots \hat{t} \dots] \rightarrow x[\dots \epsilon \dots]} \text{ (T-adapt2)}$$

2.3 An Example

Figure 5 shows an example of removal of floating point dependencies from the class `Cafe`. Though the example illustrates just the removal of fields it does highlight the various kinds of analysis that must be performed. In the example, the class instance field `innocent` has a floating point dependency and all class members containing references to this field should be removed. There are three primary analysis issues that arise:

declaration-before-use Within a given scope, a case that must be considered is when a “safe” declaration of the field `innocent` (i.e., a declaration that is supported by the JVM) occurs before any use of the field `innocent`. In this case, references to the field `innocent` occurring in this scope do not constitute a floating point dependency. An example of this is shown in the first environment method in Figure 5.

use-before-declaration In this case, a scope is encountered containing a “safe” declaration of the field `innocent`. However, a use of the field `innocent`

occurs before the declaration. Such a reference constitutes a floating point dependency. An example of this is shown in the second `environment` method in Figure 5.

ref-using-this References to the class instance field `innocent` via an expression of the form: `this.innocent`. Such references can occur from within scopes where the integer field `innocent` has been re-declared. Nevertheless, the expression `this.innocent` constitutes a floating point dependency. An example of this is shown in the `Cafe` constructor in Figure 5.

3 Infrastructure

There are a number of language independent transformation systems that could be considered for implementing the ideas described in the previous sections. Among these are Stratego [17], DMS [3], Strafunski [10], CTC [9], Maude [4], ASF+SDF [16] and HATS [20]. We used the HATS system to implement the library adaptations described in this paper.

HATS is an IDE that provides a variety of capabilities germane to transformation-based software development. These capabilities include: (1) an engine where transformation can be performed through the execution of programs written in a special purpose language called TL [21],[19], (2) a parser generator having GLR-like capabilities accepting as input extended BNF grammars together with precedence and associativity rules, (3) an abstract pretty printer, (4) graphical display facilities for viewing the structure of parse trees, (5) text editors, (6) a display showing various metrics associated with TL program execution (e.g., number of rewrites applied, etc.) and (7) some rudimentary tracing capabilities to assist in debugging transformations [22]. A dataflow diagram of HATS is shown in Figure 6.

The classic transformation scenario is to parse a source program/term, execute analysis and transformations, and pretty-print (un-parse) the resulting tree back into textual form. Here, the grammar and pretty-print rules that support these activities are language-dependent. Transformation system itself, on the other hand, is usually language-independent.

One aspect of using meta-programming systems, such as transformation systems, that oftentimes receives little mention in the literature is the breadth and depth of the effort needed to create the infrastructure that is necessary in order to make a given transformation system useable for a particular problem. In our case, we are interested in creating a domain in which it is possible to subject Java program to various kinds of analysis and manipulation. This requires,

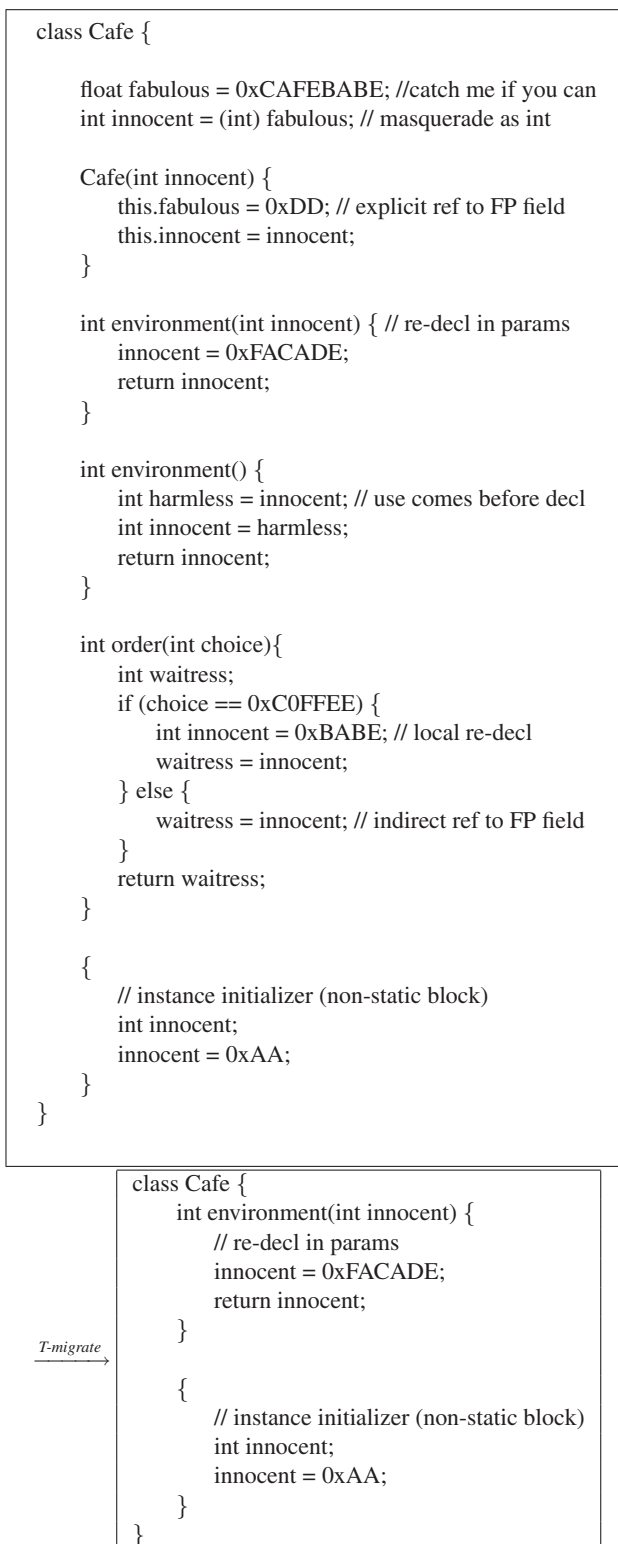


Figure 5. An example highlighting dependency analysis involved in recognizing a reference to a field.

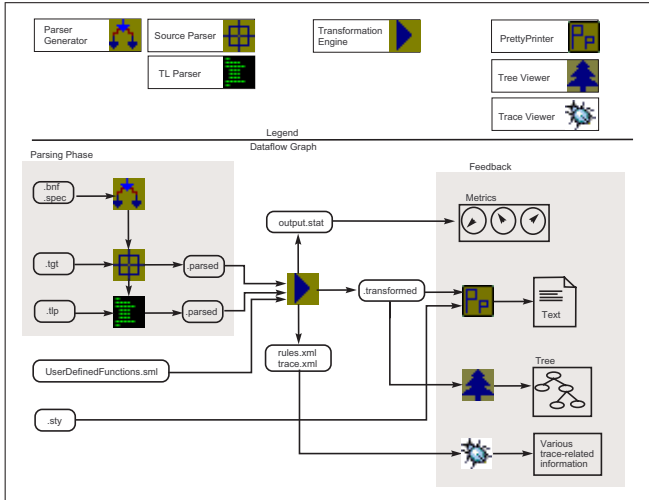


Figure 6. The Architecture of HATS from a Dataflow Perspective

at a minimum, the ability to parse and un-parse (i.e., pretty-print) Java programs.

In academic settings, the construction of parsers and un-parsers for small languages is rather straightforward. In contrast, the construction of a parser and un-parser for a real-world programming language can require substantial effort. A widely known horror story from the parsing community is the monumental effort needed to construct a parser for C++. It is not uncommon for the BNF grammar of a real-world language to exceed several hundred productions. When creating BNF grammars on this scale, the correction of even small typos in the BNF can require a significant debugging effort. Publicly available BNF descriptions oftentimes (1) contain minor typos, (2) make pseudo-implicit assumptions about the treatment of corner cases, (3) make use of extended-BNF notations, and (4) make implicit assumptions regarding the capabilities of the technology that will be used to generate a parser from the (extended)-BNF. For example, if the parser generator supports precedence and associativity rules, then a compact notation can be used to describe arithmetic expressions; otherwise, the precedence and associativity of arithmetic operations must be made explicit in the grammar.

An example of a corner case involving Java expressions arises with respect to explicit casting. In Java, the expression e can be cast to the type A by writing: $(A)e$. Furthermore, in Java, user-defined types (i.e., classes) are denoted using identifiers. Under these assumptions, unless care is taken, an expression of the form $(id)e$ will result in an ambiguous parse. For example, in the expression $(x) - y$, the symbol $-$ can be interpreted as the binary operation of subtraction. However, this expression could be interpreted dif-

ferently as an explicit cast involving the type x and the expression $-y$. Resolving issues such as these result in modifications to the grammar and its associated lexer. Such modifications increase the need to more thoroughly validate the resulting parser.

While the grammar of Java well-behaved, constructing a parser for Java was not without its own challenges. As a starting point we used the BNF given in *The Java Language Specification(3rd edition)* [7]. In order to make this BNF operational, we had to resolve several of the issues described in the previous paragraphs. In addition, we had to modify the BNF so that (most) comments could be preserved during transformation. The resulting grammar provides transformation developers explicit control over chunks of comments belonging to the portions of Java programs undergoing transformation. As a result, the output of transformations will contain all comment phrases that were not removed by the transformations.

Comments are important because they describe the intent of the source code, the history of changes to the code, and in some cases formal pre- and post-conditions of the code. Since the output of transformation for the library adaptation problem is Java source code, instead of some low-level intermediate code (where comments might be irrelevant), preservation of comments is extremely desirable. The reason being that post-transformation code inspection is part of the assurance process, and such inspections benefit greatly from the presence of comments.

After transformation, adapted source-code level programs are written to files. In HATS, the formatting performed during this stage is done by a pretty-printer which is able to format code according to the formatting style for Java outlined by Sun’s Java Code Conventions [13].

Due to the modifications made to our starting grammar [7], we conducted a test involving parsing and un-parsing (without any transformations) a batch of source files consisting of 1061 files in the following frequently used libraries: `java.lang`, `java.util`, `java.io`, `java.nio`, and `javax.swing`. The result of the test was that our Java parser was able to parse all of the code in the libraries. Furthermore, our pretty-printer frequently produced code in a more conforming format than the original source files, which we contribute to the possibility that some of the code in these libraries was written before the Code Conventions were adopted. Based on the successful testing of the grammar and the pretty-printer we concluded that we had arrived at a properly functioning grammar and we now had the necessary infrastructure for developing the transformations needed for adaptation.

4 Results

In this section we report on the results of applying the library adaptation tool on the utility package of the Java libraries – the `java.util` library (version 1.6.0). The targeted platform is the SCORE processor [11, 18]. We summarize the processor’s restrictions below:

synchronized Multi-threading and mutex features of the standard JVM is not supported. This feature of the JVM is tied to the ‘synchronized’ Java source code keyword. Thus, any mention of the keyword is removed.

floating-point operations Floating-point (FP) arithmetic is not supported. FP computations arise whenever float/double variables or methods are declared in the source code. We remove the declarations and any class members that reference such declarations.

reflection The reflection features of the standard JVM such as lookup of the run-time type of an object are not supported. This feature is related to the `Class` class and members of the `java.lang.reflect` library. We remove all dependencies on `Class` and ‘reflect’ members.

native Native methods of the libraries are implemented in platform-dependent manner; such methods are declared in library code and are implemented in non-Java code. SCORE supports some, but not all, native methods. We remove dependencies on unsupported native methods.

The ‘util’ library consists of 96 source files with the average size of a file around 1000 source lines of code (SLOC). A source file typically corresponds to one class, however there are some examples where 2 or more classes are placed in the same file (e.g. `AbstractList`). 12 classes were excluded from transformations because they fell outside of the operational profile of the SCORE platform: e.g. `Timer` is a thread-scheduling class while SCORE is a single-threaded platform, and `JapaneseImperialCalendar` is an implementation of the Japanese imperial calendar system while SCORE is intended to be used within U.S. system of measures such as Gregorian calendar.

The tool has been applied on 84 source files, 39 of which were unaffected by transformations, i.e. they did not contain unsupported features. The results of transformations of the remaining 45 classes are summarized in Figure 7.

The first two columns in Figure 7 show the original SLOC and number of class members. The column ‘Auto’

	sloc	orig	auto	man	end	sloc'
AbstCollection	429	16	1	0	15	381
AbstractList	765	22	0	0	22	769
AbstractMap	790	22	0	0	22	785
ArrayDeque	840	49	3	1	45	772
ArrayList	596	30	4	0	26	500
Arrays	4173	161	43	1	117	2788
BitSet	946	47	3	0	44	872
Calendar	2656	160	38	6	116	1619
Collection	423	15	0	1	14	357
Collections	3645	241	17	7	217	2722
Currency	393	36	7	2	27	176
Date	1315	53	45	0	8	216
DupFormFExc	52	5	0	1	4	46
EnumMap	721	66	16	2	48	484
EnumSet	420	28	12	1	15	169
EventObject	60	5	1	0	4	48
Formatter	4389	123	53	0	70	2859
GregorianCalendar	2900	82	49	0	33	745
HashMap	1038	58	14	0	44	786
HashSet	296	18	6	0	12	181
Hashtable	1112	50	3	0	47	1015
IdentityHashMap	1216	55	5	0	50	1142
IIIFormCPExc	52	5	0	1	4	46
IIIFormConvExc	68	7	1	0	6	62
InvPropFormExc	74	4	2	0	2	48
JumboEnumSet	352	21	3	0	18	295
LinkedHashMap	475	24	3	2	19	418
LinkedHashSet	155	5	4	0	1	101
LinkedList	967	52	6	0	46	888
List	569	25	0	1	24	489
Observable	197	12	0	0	12	197
PriorityQueue	721	39	7	0	32	569
Properties	1096	38	18	0	20	395
Random	532	27	19	1	7	51
RegularEnumSet	284	18	3	0	15	234
Set	357	15	0	1	14	289
SimpleTimeZone	1679	75	11	0	64	1275
Stack	125	7	0	0	7	17
TimeZone	778	44	21	2	21	283
TreeMap	2417	282	5	6	271	2289
TreeSet	523	36	2	1	33	432
UFormConvExc	52	5	0	1	4	44
UUID	489	28	5	0	23	372
Vector	1015	52	3	0	49	943
XMLUtils	191	14	9	2	3	30
Total	42343	2177	442	40	1695	29199

Figure 7. Transformation results

displays the number of class members removed using the automation. Due to some limitations described below, some members had to be manually removed to pass compilation; the column ‘Man’ lists how many members were removed manually. The final two columns provide the final number of members and SLOC.

Our transformation-based approach does not perform semantic evaluation; analysis of dependencies on unsupported features is based on strictly syntactic matching of calls to removed members. For example, if method `A.foo(int)` has been removed and `B.size()` is a supported method that returns an integer, a call to `A.foo(B.size())` will not match

the filter for `A.foo(int)`. Class members that contain such dependencies were the main reason why manual post-transformation removal was necessary.

As an example, consider class `Arrays` – the largest class in the `'util'` library with 161 members in 4173 SLOC. This class contains various methods for manipulating arrays such as sorting and searching. The 43 removed members in this class depended on `float/double`'s and methods in the `Class` class. Among the removed members was the method `deepToString(Object[],StringBulider,Set)` that returns a `StringBulider` representation of a given `Object` array with no duplicate elements. This method was called from within `deepToString(Object[])` with the actual parameters `(Object[],StringBulider,HashSet)`; note that `HashSet` is a subclass of `Set`. Since the formal and actual parameters did not match syntactically, the calling method was not removed by transformations and had to be removed manually. The combined auto and manual removals produced 117 remaining members in 2788 SLOC.

The task of manual inspection of transformation results was eased due to the presence of a compiler, which would report an error whenever transformations had not removed sufficient number of class members due to the syntactic limitation described above. The compiler served both as a helper for tracking the missed dependencies (e.g. source code line numbers of a missed member) and as an assurance of correctness. Transformations guarantee that direct dependencies on unsupported features (e.g. `'float x;'`) are removed. The compiler guarantees that any transitive dependencies on removed members are also removed. The two tools together provide a full coverage of unsupported features.

Statistically, transformations removed 442 out of total 482 class members – a 91% coverage of all unsupported members; on average, less than 1 member per class was removed manually (40 members in 45 classes). The total number of members reduced from 2177 to 1695 – 23% reduction; SLOC went from 42343 to 29199 – 38% reduction.

Overall, our experience with `java.util` suggests that automated transformations provide a significant help. The manual post-transformation member removal took about a day, instead of possibly weeks of un-aided manual library adaptation. The industrial rule-of-thumb, as suggested by Akers et al. [1], states that anything over 75% adaptation rate using automated techniques produces significant cost savings over the manual approach. Using the transformations-based approach presented here, we have achieved a 91% adaptation rate.

5 Related and Future Work

Majority of the work on transformation of Java code utilizes behavior-preserving refactoring techniques [14] [2][5].

Refactoring is a transformation that alters structure of existing code without affecting its observable behavior. Refactoring can range from simple modifications like formatting code with proper indentations to more complex modifications like converting blocks of code into sub-routines or methods. Refactoring has been used to customize and optimize Java code including Java libraries. Sutter et al. [14] profile libraries and based on their execution speed and memory footprint create optimized versions of the standard library 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 [8].

Recent developments in the use of refactoring to migrate Java code use semantic type information and type constraints. Here, the typing rules of Java are captured as type constraints that have to be satisfied by refactored code. The impressive work by Tip et al. [15] uses type constraints to preserve behavior in extraction of interfaces and pull-up of members into a super-class. Refactoring with type constraints has also been used to remove deprecated code [2] and convert legacy code with downcasts into generic code that eliminates downcasts [6].

The work presented here differs from other work related to Java libraries in that we modify the libraries directly instead of 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 Sun Microsystems does not provide a migrated version of the standard libraries for use on their Java Card platform [12].

The chief limitation of this work is absence of complete type information; we track strictly syntactical dependencies. Future work entails inclusion of type information that can be gathered from superclass-subclass hierarchies and class members. Essentially, this involves scanning of class hierarchies in a top-down manner collecting member type signatures into a set of filters. Then, the fully scoped semantic dependency on unsupported features can be precisely calculated eliminating the need for post-transformation manual revision of code entirely.

6 Conclusion

Library adaptation is an essential part of providing high-level language support for embedded system programmers. Libraries such as `java.util` are evolving artifacts and periodically new versions of these libraries are released. As a result, in order to remain current the adaptation of a particular library must be repeated in response to a new release of the library. In this paper, we presented a transformation-based approach for Java library adaptation. The scope of

the adaptation effort is currently limited to the removal of portions of the library dependent upon features that are unsupported by a target JVM. Syntactic and semantic well-formedness constraints influence the shape of terms that are candidates for removal. In particular, these constraints lead to a simplifying generalization that limits the scope of removal to a class' member terms (e.g., fields, methods, constructors, inner classes, and class blocks).

Declarative-style rules are given describing the adaptation process. The declarative rules *T-transitive* and *T-migrate* have premise conditions that require semantic analysis (e.g., the recognition of \hat{t} in a particular context). This analysis goes beyond syntactic matching capabilities and can be summarized as a decision procedure for recognizing references to unsupported elements within a class (e.g., floating point dependencies).

References

- [1] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49:275–291, 2007.
- [2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring Support for Class Library Migration. In *Proceedings of OOPSLA 2005*, pages 265–279, San Diego, California, United States, 2005. ACM.
- [3] I. Baxter, P. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 625 – 634, 2004.
- [4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
- [5] D. Dig and R. Johnson. The Role of Refactorings in API Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389 – 398. IEEE, 2005.
- [6] A. Donovan, A. Kiezun, M. S. Tschantz, and M. D. Ernst. Converting Java Programs to Use Generic Libraries. In *Proceedings of OOPSLA 2004*, pages 15 – 34, Vancouver, BC, Canada, 2004.
- [7] J. Gosling, B. Joy, G. L. S. Jr., and G. Bracha, editors. *The Java Language Specification(3rd edition)*. Sun Microsystems Press, 2006.
- [8] J. Henkel and A. 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.
- [9] G. Kniesel. A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006. ISSN 0944-8535.
- [10] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.
- [11] J. A. McCoy. An Embedded System for Safe, Secure and Reliable Execution of High Consequence Software. In *HASE 2004: The 5th IEEE International Symposium on High Assurance Systems Engineering*, pages 107–114, Albuquerque, New Mexico, United States, 2004. IEEE.
- [12] S. Microsystems. Java card technology. <http://java.sun.com/products/javacard/>.
- [13] S. Microsystems. Java code conventions. <http://java.sun.com/docs/codeconv/codeconventions.pdf>.
- [14] B. D. Sutter, F. Tip, and J. Dolby. Customization of Java Library Classes using Type Constraints and Profile Information. In *Proceedings of ECOOP 2004*, pages 585 – 609, Oslo, Norway, 2004.
- [15] F. Tip, A. Kiezun, and D. Baumer. Refactoring for Generalization using Type Constraints. In *Proceedings of OOPSLA 2003*, pages 13–26, Anaheim, California, USA, 2003.
- [16] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.
- [17] E. Visser, Z. e. A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
- [18] 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 8th IEEE International Symposium on High Assurance Systems Engineering*, pages 167–177, Tampa, Florida, United States, 2004. IEEE.
- [19] V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.
- [20] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.
- [21] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
- [22] V. L. Winter, C. Scalzo, A. Jain, B. Kucera, and A. Mametjanov. Comprehension of Generative Techniques. In *Software Transformation Systems Workshop (STS)*, 2006.