An Intelligent CASE Tool for Porting Mobile Java Applications

Ioannis T. Christou[1], Sofoklis Efremidis[1], Aikaterini Roukounaki[1],and Marios Anapliotis[2]
1 Athens Information Technology, 19.5 km. Markopoulou Ave.
19002 Peania, Greece
{ichr,sefr,arou}@ait.edu.gr
WWW home page: http://www.ait.edu.gr
2 INTRALOT S.A., 64 Kifissias Ave., Maroussi 15125, Greece
anapliotis@intralot.com
WWW home page: http://www.intralot.com

Abstract. Today, many mobile device vendors offer their own versions of the Connected (Limited) Device Configuration (CLDC 1) and Mobile Information Device Profile (MIDP 2, 3) of the Java 2 Mobile Edition (J2ME 4). In addition, depending on the device characteristics they offer device-specific or series-specific libraries and APIs extending or complementing those specified in the standard CLDC and MIDP. As a result, porting a Java application written for one device to another is often a very tedious and time-consuming task for the developers. We present *SeqFinder*, an intelligent CASE tool for assisting the porting of Java mobile applications. *SeqFinder* eases the porting task by automatically generating all minimal method invocation sequences that lead to an object of a specific type, thus relieving the programmer of the effort to manually search the manufacturer-provided SDK Java archives to find how to accomplish a particular task, for example, how to initiate data transfer through a socket or an HTTP connection.

1 Introduction

It is an unfortunate fact that the promise of Java 5 as a "write once, run everywhere" language proved to be overly optimistic in the mobile phone market-place. This is by no means a criticism on the language itself, as Java is probably the best-suited language for writing applications for mobile phones and mobile information devices in general, but rather a realization that the extreme variation in those devices' characteristics necessitated different versions of the language specifically tailored to these devices. Undoubtedly, the area with the most variations in the offered APIs is the Graphical User Interface, but as different devices support different features and

offer the corresponding APIs for developers to take advantage of, there is a wide range of capabilities of different devices and, correspondingly, of the APIs supporting them.

Less known is the fact that even standardized APIs such as the connectivity API that allows access to the Internet and servers and services available over HTTP or socket protocols are often poorly implemented by the device manufacturers and cause serious porting problems to application developers. As an example, consider the trivial standard code snippet for connecting from a Mobile Information Device applet (MIDlet) via HTTP to an HTTP server.

```
// mMessageItem defined before
HttpConnection hc = null;
InputStream in = null;
String url = getAppProperty("URL");
try {
  hc = (HttpConnection)Connector.open(url);
  in = hc.openInputStream();
  int contentLength = (int)hc.getLength();
  byte[] raw = new byte[contentLength];
  int length = in.read(raw);
  in.close();
  hc.close();
  // Show response to the user
  String s = new String(raw, 0, length);
  mMessageItem.setText(s);
}
catch (IOException ioe) {
  mMessageItem.setText(ioe.toString());
}
```

In certain phone models the previous code will not work, as their virtual machine does not implement the `openInputStream` method of the `HttpConnection` class that returns `null` instead, effectively not conforming to the MIDP specification. Instead, the method `openDataInputStream` is implemented, which works as expected.

To further complicate things, many (early) phone models that claimed to support the MIDP specification had serious bugs in the built-in Kilobyte Virtual Machine (KVM) especially regarding synchronization issues, thread management, and memory management. These defects combined make writing complex applications for such devices at times a daunting task.

Porting of existing applications from one device type to another (of the same or different vendor) is an equally complicated process. The particularities of the APIs as well as the KVM and MIDP implementations of the concerned device types have to be taken into account, making porting a time consuming and tedious process. Typically, application developers and/or application porters need to try several alternative sequences of object constructions and method invocations for establishing a successful connection. Any tools that could assist to the (semi)automatic

generation of such alternative code fragments will be of immense help to the application porters.

To reduce developer time for porting applications to a new mobile device type we have developed *SeqFinder*, a tool that automatically generates all possible code sequences that lead to a specified target, i.e., an object of a certain type as specified by the developer, for example the initiation of a successful data transfer. Note that by code sequences we mean sequences of constructor and method invocations and/or retrievals of field values. The tool generates a separate MIDlet for each sequence of Java statements that leads to a reference to such an object. Each sequence element is either a declaration and initialization of a primitive type such as `int i1 = 0;` or a (possibly static) method invocation such as

```
Connection o1 = Connector.open(s1, i1, b1);
```

For any method invocation that requires references to class objects, the generated sequence includes a valid such reference in a previous line. This is accomplished by recursively generating objects of the type of the calling object and the parameters of the target method.

The automatic generation of all such sequences and their embedding in appropriately wrapped Java code so as to be compile-able and testable MIDlets can help the developer quickly find how to use the APIs provided by the manufacturer in order to accomplish a particular task, such as reading data from the network or sending data to a server.

1.1 Related Work

Several tools and environments for targeting of applications for various families of mobile devices have been proposed and a number of them are in widespread use. All of them focus on the targeting chores from a given code base but do not provide support for developing such a code base at first place.

J2MEPolish 6, 7 is a freely available, powerful toolset that heavily relies on a database of devices containing their specific characteristics, which are taken into account in the targeting process. The mechanism used for targeting is based on the same kind of pre-processing as C. The approach taken by J2MEPolish is feature-driven, i.e., application targeting is governed as specified by the pre-processing directives that take into account the characteristics of the device of interest as recorded in the device database. Otherwise, the tools do not provide functionalities for dynamically developing sequences of "winning" method invocations which will lead to a successful target, e.g., successful transfer of data over an HTTP connection.

The j2me-device-db 8 project started out as a project aiming to create a database of mobile phone characteristics and even bugs, but developed to a set of tools for developing mobile applications via the use of pre-processor directives very similar in nature to J2MEPolish.

Tira Jump 9 (latest release 3) is another software solution that simplifies project planning and, in addition, enables the efficient deployment of content across a range of diverse mobile devices. It employs an extensive knowledge-base that registers most device specificities. The integrated workflow and robust digital asset management systems create an end-to-end solution that simplifies the process, enhances consistency, quality and controls mobile content deployment on a global

scale. The Jump Developer Desktop (JDD) manages the adaptation and optimization of reference applications to support new handsets. The tool is available in two versions:

- The JDD Java Edition (based on Eclipse) provides the key interface to the Jump Transformation Engine (JTE) when adapting Java applications.
- The JDD BREW Edition is a Microsoft Visual Studio .NET add-in and adapts BREW 10 applications.

Tira Jump 3 is built around the concept that between any two mobile devices there exists a finite set of differences. Consequently, the platform makes use of a reference device and contains, maintains and updates differences between every supported target device and the reference device. For each difference it provides a series of adaptation instructions to convert content from one device format to the other.

The Jump Transformation Server supplies the device plug-ins and performs the conversion of the Java or BREW application or the ringtone, screensaver or wallpaper. Similar to J2MEPolish, the Tira Jump tools allow for targeting of mobile applications to a range of mobile devices but do not provide methodological support for developing sequences of "winning" method invocations leading to a successful target.

Relevant to the *SeqFinder* algorithm development is perhaps work on hyper-graph shortest paths 11, for indeed it is possible to model the problem domain as a hypergraph in the following way: every type in the SDK of the mobile device can be considered as a node, and every public method or field of the SDK can be considered as a Back-feed arc connecting the type of the class declaring the method and the types of each argument of the method to the type of the object the method returns. The problem is then one of computing all minimal different paths of reaching a desired object in this graph starting from primitive types. This model does not take into account inheritance and polymorphism issues but may yield fruitful results in the near future.

2 System Description

SeqFinder operates on a data repository (e.g., a relational database) that contains all the necessary information about the classes that are available on a specific mobile device. *SeqFinder* takes as input the type of an object that the developer wishes to construct and returns all minimal code sequences targeted for this specific device that produce such a reference in the format of valid Java code that can be automatically compiled and executed for testing purposes. The generated Java code sequences contain tool-default values for variables and arguments of primitive types and for the construction of any intermediate objects.

These sequences are fed into a specially designed GUI for fine tuning by the developer. The GUI allows the developer to quickly and intuitively interact with each of the above sequences to fill out the placeholders that contain the generated tool-default values and, as a result, to generate a set of Java files that can be executed directly on the Java-enabled mobile device. Among the generated Java programs

those that succeed in successfully transferring data though a connection are tagged as the "winners".

In this section, we present a recursive algorithm that produces all possible sequences of method invocations that, eventually, result in an object of a given type starting from primitive Java type variables.

Input to the Algorithm: The algorithm presented below takes as input

- the type of an object to be constructed by a sequence of method invocations, and
- a unique key that identifies the devices, at which these code sequences are targeted.

Output of the Algorithm: Vector of all minimal distinct sequences of steps leading to a reference to an object of the requested type (specified target by the developer). The produced reference can be of the requested type or of any of its subtypes or its supertypes (in this case casting is necessary). These sequences are invariant with respect to values given to primitive types (e.g., int variables' values do not matter in this program, and each primitive type is assumed to have a unique default value to be assigned whenever such a variable is created). To avoid looking for all the infinite sequences of code segments that arise when multiple objects from the same type can be constructed, we introduce the constraint that only one object of each non-primitive type can be constructed in any given sequence. We also ignore any methods or fields whose type belongs in the same class hierarchy with the class, in which they belong (i.e. is any of the supertypes or subtypes of this specific type).

Finally, *the developers are able to impose two kinds of limitations on the generated code sequences. First, they can limit the maximum length of the code sequences that SeqFinder generates. Second, they can exclude certain packages, classes and members from the generated code sequences*, meaning that they can force SeqFinder to ignore any member that involves in any way the use of these packages, classes of members.

2.1 Auxiliary Data Structures

Our algorithm makes use of auxiliary data structures and methods as described in this subsection.

1. A method `newName` is provided that receives the name of a type (e.g. `int` or `java.lang.Integer`) as a parameter and returns
   a. the argument with the '1' appended to it, if the type is not a primitive type
   b. a new (unique) name, otherwise.
2. Java source-code-level object definitions and initializations of the form

```
T o1 = (TT) o2.m(a1, .., an);
```

are represented by the following data structure:

```
Command {
  String objType;
  String objName;
  String realType;
  String CCOName;     // called class obj. name
  String CCType;      // called class type
```

```
    String memberName;  // called class member name
    Vector<Parameter> paramList;
    // paramList has the name-type pairs of method's
args
  }
```

where

a. type T is represented by `objType`,
b. object o1 is represented by `objName`,
c. type TT is represented by `realType`
d. object o2 is represented by `CCOName`,
e. the type of o2 is represented by `CCType`,
f. method m is represented by `memberName`, and
g. the names and types of the parameters are represented by `paramList`.

In case method `m` is a static method, the value of `CCOName` member is equal to `CCType`. If m is a field and not a method, then `paramList` is `null`. If `m` is a constructor, then `CCOName` is null.

3. Method descriptions are represented by the following data structure

```
CommandStruct {
  boolean isStatic;
  String type;
  String CCType;
  String memberName;  // type of each method parameter
  Vector<String class_type> PTypes;
}
```

where

a. `isStatic` is true, if the member is static,
b. `type` is the member's type,
c. `CCType` is the type of the object on which the method is invoked or whose field is accessed,
d. `memberName` is the name of the member,
e. `PTypes` is the types of the parameters

In case the member is actually a field, `PTypes` is null.

4. A method `findAllMembersOfType` is provided that, given a type C, returns *all* members (fields, methods, constructors) of classes contained in the data repository that return an object of the requested type or of any of its supertypes or subtypes. This method returns also members whose type is an array type with elements of the required type.

5. A method `findIncomplete` is provided that, given a vector of `Commands` `v` and an initially empty vector of strings `tv`, returns the first `Command` object in `v` that is incomplete. An incomplete `Command` is one, whose called object or one of its parameters have not already been defined in any of the `Command` objects that appear before `v` is not of primitive type or string This is determined by searching through the `Command` objects before `v` for an `objType` that belongs to the class hierarchy of the type of the specific object.

Effectively, the `findIncomplete` method identifies the first use of a yet undefined object in the sequence of `Commands` under construction; such an object

has to be eventually defined and initialized. The method also fills vector `tv` with two strings, the first is the type name of the object that has been identified in `v` (the value of `CCType` in `Command` or the value of the `PType` member in the `Pair<., .>` in `paramList`), and the second is the name of the object itself (the value of `CCOName` or `PName` in the `Pair<., .>`). If no Command object has a name in the command that is not defined below in vector `v`, then `null` is returned.

3 The Generator Algorithm

The output of the algorithm is a *vector of vectors of* `Commands`. Each one of these vectors of commands describes a sequence of steps that lead to an object of the requested type. The algorithm assumes that the user may have specified a set of packages, classes and members that should be ignored, as well as the maximum length (maximumLength) of the generated code sequences.

```
vector SEQ(Commands, ObjectType, ObjectName) {
  CommandStructs=findAllMembersOfType (ObjectType);
  create new empty vector called result;
  foreach CommandStruct in CommandStructs {
    if CommandStruct has excluded packages,
            clases or members
      continue;
    create the corresponding Command c;
   create a new vector v containing all the Commands + c;
   add v to the result;
  }
 foreach v in result {
   incompleteCommand=findIncomplete (v,tv);
   if incompleteCommand is NOT null {
     if v.size == maximumLength {
       remove v from result;
       do not move forward in result;
     }
     type = tv[0];
      name = tv[1];
     result' = SEQ(v, type, name);
     if result' is NOT empty {
        replace v in result with result';
        do not move forward in result;
      }
    }
  }
  return result;
}
```

Each Command object represents a Java statement of the form `int i1 = 0;` or more complex statements such as

```
java.util.Hashtable java_util_Hashtable1 =
                        new java.util.Hashtable();
```

or `A A1=SomeObjectRef.someMethod(C c1);` and so on. Static methods of the form

```
javax.microedition.io.Connection
 javax_microedition_io_Connection1=
  javax.microedition.io.Connector.open(s1);
```

are of course also expressible.

A `Command` object captures in its `objType` data member the name of the class `A` in the statement above. The name of the object reference `A1` is captured in `objName`. The name of the type of the object reference `SomeObjectRef` is captured in `CCType`, while the name `SomeObjectRef` itself is stored in `CCOName`. The name of the method invocation `someMethod` is stored in the `memberName` data member and similarly the name of the type and reference of each argument is stored in the data member `paramList`.

The algorithm accepts as an input a partially incomplete sequence of `Command`s and it attempts to locate all the possible ways to complete it (in terms of the specific object type that it receives as a parameter). For this reason, it locates all Java statements that return an object of the desired type (or of any of its supertypes or subtypes) and creates a separate vector of `Command`s for each one of them, which consists of the received code sequence followed by the new `Command`. The algorithm then computes recursively for each name that appears in each of the newly generated code sequences and that has not been defined already in this sequence all code sequences that will define this name, and prefixes them with the current sequence. It then replaces the current sequence in the result vector with the expanded code sequences, resulting in all code sequences that can be used to obtain a reference to the requested type.

It is important to notice the way `newName(type)` works according to its specification. This method always returns the same name when the input is a non-primitive type. This limits the code sequences constructed to sequences that create only one object for each different type considered. This is not a serious limitation of the program and it avoids the otherwise almost inevitable infinitude of different possible code sequences. To clarify why this is the case consider two classes `A` and `B`. Class `A` has a unique constructor `A(B b)` and there is no other way in the program to obtain a reference to an `A` object. Class `B` on the other hand has two constructors, `B(int i)` and `B(A a)`. Now if we want to find all code sequences leading to an `A` object, the last line of each such sequence has to be `A A1 = new A(B1)` where `B1` is an object of type `B`. Apparently, there are two ways to obtain an object `B`. One is by having a statement of the form `B B1 = new B(i);` where `i` is an int variable and the other is by having a statement of the form `B B1 = new B(A2);` where `A2` is an object of type `A`. If the name `A2` is not "unified" with `A1`, then we must find all ways of creating an `A` object again and add them to our current code sequences, which leads to infinitely long object creations of the following form in reverse:

```
A A1 = new A(B1);
B B1 = new B(A2);
```

```
A A2 = new A(B2);
B B2 = new B(A3);
...
```

To further simplify the things, *SeqFinder allows only for one object from any class hierarchy to exist in a sequence.* In other words, if an object of class A has been created in a Command of a specific command sequence, then no other object of this specific type or of any of its subtypes or supertypes will be created in this code sequence. The already created object can be used in all of the above mentioned cases.

4 Computational Results

As shown in Table 1, the system is capable of finding all valid sequences of 2 or less method invocations leading to a `java.io.InputStream` object reference in less than 16 minutes on a commodity MACBook laptop running at 2Ghz the MAC OS X Operating System and Java JDK 1.6. The response times, even though they are not real-time, still present huge time-savings to developers who previously had to spend *more than two days* in order to make a single network-centric Poker game to establish connections and exchange data with the game server. It is also interesting to note how the number of different code sequences for obtaining a `Connection` object reference jumps from 35 in the case of two-step sequences to 201 different sequences of length 3. The length 3 sequences of course include the length 2 sequences. The table also shows some other first results of running the system.

Table 1. Experimental Results.

Object Type	Number of Steps	Response Time	Number of Sequences
`java.io.InputStream`	2	15 min	164
`java.io.DataInputStream`	2	16 min	292
`Javax.microedition.io.Connection`	2	5 min	35
`Javax.microedition.io.Connection`	3	6 min	201

5 Conclusions and Future Directions

We presented *SeqFinder*, a tool that automatically generates all sequences of Java method invocations leading to an object reference of a given type (or any of its sub-types). This tool helps mobile application porting and development tasks as it relieves its user (application developer) from the cumbersome task of having to search a manufacturer-specific APIs for functionality described in possibly non-existing or just poor java-docs. It does so by emulating the actual process by which developers approach this task (bottom-up), i.e., by letting the user automatically search for all classes that support a “connect” method, or to directly find all possible

ways to get a `java.io.OutputStream` object that can use to send data through the network.

In certain cases, the number of sequences generated can be overwhelming. In addition, the user currently has to know (or guess) what values to assign to the primitive data types variables. For these reasons, we are currently experimenting with machine learning techniques to offer a more intelligent user-interface that will help the user choose quickly the code sequence (by sorting them in a "likelihood-index") and primitive values combination that are most likely to be successful at performing a particular task. This sorting would be performed by examining previous test sequences generated and tested. The tests would then have resulted in "success" or "failure" which, if recorded, can lead to a training set of labeled examples. Standard supervised classification techniques can then be used to infer the likelihood of a proposed sequence to be a winning sequence and from this information any vector of all possible sequences can be "intelligently" sorted.

References

1. Connected Limited Device Configuration (CLDC), JSR 30, JSR 139, http://java.sun.com/products/cldc/
2. Mobile Information Device Profile (MIDP), JSR 37, JSR 138, http://java.sun.com/products/midp/
3. Ortiz, C.E. and Giguere, E., Mobile Information Device Profile for Java 2 Micro Edition: The ultimate guide to creating applications for wireless devices, John Wiley and Sons, Inc., New York, NY, 2001.
4. The Java ME Platform, http://java.sun.com/javame/index.jsp
5. Gosling, J., Joy, B., Steele, G., and Bracha, G., The Java Language Specification, Second Edition, Addison-Wesley, Reading, MA, 2000.
6. Virkus, R., Pro J2ME Polish: Open Source Wireless Java Tools Suite, Apress, Berkeley, CA, 2005.
7. J2MEPolish, http://www.j2mepolish.org/
8. The j2me-device-db project. http://j2me-device-db.sourceforge.net/pmwiki/
9. Tira Jump 3, http://www.tirawireless.com/
10. Binary Runtime Environment for Wireless, http://brewx.qualcomm.com/
11. Gallo, G., Longo, G., Nguyen, S., and Pallottino, S., Directed Hypergraphs and Applications. Discrete Applied Mathematics, 42:2–3, 177–201, 1993.