Debugging Tools for MIDP Java Devices

Olli Kallioinen[1] and Tommi Mikkonen[2]

[1] Sasken Finland, Tampere, Finland
olli.kallioinen@sasken.com
[2] Tampere University of Technology, Tampere, Finland
tommi.mikkonen@tut.fi

Abstract. Mobile Java development using CLDC and MIDP can be very restricting, not only because of the more restricted libraries and older Java language, but also because some very basic development tools are not available in many situations. One of the biggest problems when debugging a midlet – a CLDC/MIDP application – is that when running a mobile Java application in a real device, stack traces are not available. Also other tools, like profiling tools, only work in certain emulators. In this paper, a set of improved tools for mobile Java development is introduced. Instrumentation, a well-known technique is used to work around the restrictions of the Java sandbox. Consequently no special support is required from the platform.

Keywords: Mobile Java, debugging.

1 Introduction

Sun's mobile Java platform (Java Micro Edition, Java ME) has not been a very hot topic lately. New, advanced mobile devices with platforms like Android, Maemo, and iPhone have been stealing most of the media attention. However, when comparing mobile devices that are able to run third party software, Java ME is still by far the most common mobile platform at the moment.

One of the most frustrating problems when developing application targeted to Java ME devices, is that stack traces are often unavailable. Stack traces are normally used to locate the cause of run-time errors. The traces are printed to the standard system out stream that is usually not available when running a program on a real device. In Java SE it is possible to redirect the stream to any desired destination [1], but this redirection possibility was left out from Java ME [7]. It would not be a big problem if it would be possible to access the traces programmatically, like in Java 1.4, but this is not available either. This problem describes well how the combination of small shortcomings in Java ME prevent some very basic functionality that most developers take for granted in modern environments. Also other tools like profilers usually work only in an emulator.

This paper proposes better tools for Java ME development. The practice of manipulating Java binaries after compilation is used to implement the tools. This method allows all the tools to be used regardless of the environment where the application is running. The tools can be used to find the cause of a problem in situations where the emulator cannot be used. For example a problem could only occur on certain devices, or in some cases an application cannot be run on an emulator at all.

The rest of the paper is structured as follows. Section 2 discusses mobile Java and Java language internal workings.Section 3 describes improvements and the principles that have been used to implement them. Section 4 evaluates the implemented tools and describes how they can be used. Section 5 summarizes the presented improvements..

2 Java Micro Edition

As the Java platform was growing it was split into multiple editions. In addition to the Standard Edition (Java SE) two other editions were created: Enterprise Edition (Java EE) for application servers, and Micro Edition (Java ME) for devices with limited resources. Java ME is further divided to smaller parts to be able to support a very heterogeneous set of devices. Each Java ME runtime environment consists of three parts: (1) a *configuration* that defines the set of basic libraries and virtual machine capabilities, (2) a *profile* that defines a common set of APIs for a smaller group of similar devices, (3) a set of *optional packages* for other specific technologies like Bluetooth or SMS.

Two different configurations are currently available: *Connected Limited Device Configuration* (CLDC), and *Connected Device Configuration* (CDC). Multiple profiles exist, but in general *Mobile Independent Device Profile* (MIDP) is the most commonly used one. The combination of CLDC and MIDP, which is targeted for low-end devices like feature phones, is one of the most common runtime environments in existence, with billions deployed in different mobile phones. A Java ME application built using CLDC and MIDP is called a *MIDlet*.

2.1 Mobile Java Development Using CLDC and MIDP

There is a variety of different kinds of tools available for a Java developer. Multiple different Java specific tools can be used for writing code, compiling, packaging, debugging, testing, static analysis of source code, and so on. Some of these tools can also be used to develop mobile Java applications, but the restrictions of CLDC also prevent the use of many tools.

The build process for building a MIDP application has some differences compared to building a normal Java SE application. In addition to the normal Java Development Kit (JDK), the Java Platform Micro Edition Software Development Kit (Java ME SDK) is required. The standard Java compiler can be used, but the Java source code version and target class file version need to be set appropriately, as the newest class file format supported by CLDC is the JDK 1.4 class format. The Java standard libraries must also be replaced with the CLDC ones. The standard Java compiler supports cross-compiling, meaning that classes can be compiled against bootstrap libraries instead of the normal JDK class libraries and the target class file format can be set to other JDK versions [4].

In comparison to the class format of Java 1.3 that CLDC is based on, an extra step is added to the build process [7]. The classes need to be *preverified* before the classes can be used by the CLDC virtual machine. The CLDC virtual machine uses a simplified process to verify the correctness of classes to make loading faster and to save memory, and consequently classes must be preverified before they can be loaded. The Java ME SDK provides a preverification tool that is usually run after compiling the classes.

Finally, compiled and preverified classes need to packaged into a JAR package before the MIDlets in them can be installed and run. Information about the MIDlet must be included in the manifest file inside the JAR file. Furthermore, the JAR file is usually accompanied by a JAD file that describes the MIDlet properties [10].

2.2 Restrictions When Developing MIDlets

When developing for mobile devices many special considerations must be taken into account. Devices usually have less resources like memory and storage space than in a full Java SE desktop environment and the CPU of the device might not be very powerful. One further speciality is the user interface. Typically only one MIDlet can be running at a time and there is no standard support for inter-MIDlet communication. Also, in many platforms MIDlets cannot be running on the background while the phone is running its native applications.

From developers point of view, the most notable difference between Java SE and Java ME is that a lot of classes have been dropped from the basic Java library and some functionalities have been replaced with Java ME specific classes. In MIDP the AWT and Swing UI libraries are replaced with the LCDUI library [8]. In addition, many language features have been dropped from CLDC. Furthermore, all the newer features of the Java language are missing.

Same debugging tools can be used in Java ME applications as in Java SE. When using an emulator, all that needs to be done is to launch the emulator in debugging mode and attach it to a debugger. Debugging with a real Java ME device (on-device debugging) requires specific support from the device. Device specific differences are common, and on many platforms no support is provided. In reality, using a debugger on an actual device is usually cumbersome, slow or not possible at all. Often, logging to a COM port or to log files is the best available solution for finding the cause for issues in the developed MIDlet.

Exceptions and stack traces are good tools for finding issues in Java applications. The restrictions of CLDC limit the use of stack traces, however. When an exception is thrown in Java SE, it is easy to locate the cause of the exception by following the stack trace attached to the exception. In CLDC it is only possible to print the traces to standard system output using *Throwable.printStackTrace()* [7]. This makes it hard to get the traces when using a real device as the standard output may not be accessible. Some devices provide a way to read the standard output prints, but such features are device and platform specific.

In Java SE it is possible to access the stack trace information programmatically [3], but the API is not available in CLDC. Another Java SE option that is not available in CLDC is to reassign the *PrintStream* used by *System.out* using *System.setOut()* [2]. Finally, setting a default exception handler for threads is not possible in CLDC.

Depending on the emulator that is used, it is usually possible to configure the emulator to show exceptions implicitly. Some vendor specific tools may be used to get the traces even on actual devices but the tools are in many cases not publicly available. In some cases a MIDlet can use proprietary functionality that is not available on the emulator and the only possibility is to run it in a real device. In these cases the exception traces provided by the emulator are again unavailable. Even if the developer can access the stack traces, they may not include the source code line numbers like Java SE traces do. In fact, the debug information is not used by the reference CLDC virtual machine implementation even if the source line debug information is available in the classes.

The lack of stack traces when running an application in a real device becomes even a bigger issue because of another Java ME related problem: *fragmentation*, caused by the huge number of different Java ME devices that have different kind of hardware, operating system, and virtual machine implementation [5]. It is common to have an application that runs on an emulator and on some devices, but crashes in some other phone models. Finding out what is wrong without proper stack traces can be a time consuming operation. Usually debugging prints need to be added to pinpoint the problem source and each time the application needs to be built, packaged, installed, and restarted on the device. All the information needed to locate the exception source are in stack traces.

The sandbox of Java ME is much more restricted in comparison to Java SE. Due to Java ME restrictions, it is generally not possible to extend the functionalities that are offered by the platform.

2.3 Java Virtual Machine and Bytecode

The virtual machine and its instruction set resemble real hardware and the instruction set of a real hardware machine. Especially the instructions that are used to manipulate memory and for performing arithmetic operations are similar. In addition, some higher level concepts have been also introduced. For example, the virtual machine directly supports objects and exception handling [6].

When Java source code is compiled, it is compiled to a hardware-independent binary format called the *class file format*. Although it is called a file format, and usually files are used, the binaries do not necessarily need to be stored to files. Each class file represents the Java class or interface it was compiled from. The file format also addresses details like byte ordering to achieve platform independence. The instruction set of the virtual machine is used to represent the Java source code. In the class format, each instruction is denoted by one byte, allowing 256 different instructions. That is also why term *bytecode* is often used to describe the compiled instructions. The bytecode instructions take only a small part of typical class files, as other information, like the symbol table containing literals, need to be also included in each class file. The file format and the instruction set are designed to be used with the Java language, so the supported features match closely to the features of the Java language [6].

The class files preserve the basic structure of the source code classes. The names of packages, classes, methods and fields are also preserved in the class file. All the information to reference a class is available in the compiled Java class so it is possible to reference the class from Java source code, even if the source code for the referenced class is not available. This enables decompilation of Java class binaries so that the produced decompiled source code is close to the original source code. Even local variables names can be preserved if the information was included in the class files when it was compiled [6].

The virtual machine specification only specifies the abstract virtual machine. Many implementation details can be defined by the virtual machine implementation. For example, the used garbage collection algorithm is virtual machine specific. Similarly, the bytecode could be interpreted at runtime, compiled to native code, or even run directly on hardware. When interpreting bytecode, just-in-time (JIT) compilation is often used to improve performance. This means that when the bytecode is being executed, some often called classes may be further compiled to native machine code.

3 Improving the Java ME Toolset

When custom class loaders are available, class bytecode can be modified at run-time. A custom class loader can be defined that alters the bytecode of the class when it is being loaded into memory. Java 5.0 has a specialized *java.lang.instrument* package that provides built-in support for modifying the bytecode of classes when they are being loaded [3]. Modifying an application to gather information about the execution of the application is often called *instrumenting* the bytecode or just *instrumentation* [9]. This kind of dynamic runtime modification is not possible in CLDC. There is no support for custom class loaders either [7]. The only available possibility to modify the bytecode, is to do it statically after compiling, before creating the final MIDlet JAR package. The modified classes also need to be preverified again before packaging.

Modifying bytecode introduces the possibility of adding new errors to otherwise working code. Modifications can be checked already on build-time and the instrumentation libraries have support for verifying that the modification produces valid bytecode. Such problems are usually found when loading and verifying the class, or when preverifying a class in CLDC. If the aim of the bytecode modifications is to change the functionality of the application, possibly introduced problems can be very cryptic and hard to debug. Even if the application is just instrumented to gather information, errors in the instrumentation can cause the functionality of the application to change.

The basic idea of all the improvements this paper is modifying the compiled Java bytecode before creating the final distributables. As the Java class format has no direct connection to Java language, also constructs that would not be possible in normal Java language can be used in valid Java class files.

3.1 Tracing Method Calls

The basic idea of tracing method calls is that a piece of Java bytecode is added to the beginning and end of each method. This makes it possible to receive a callback every time a method is entered or exited, which in turn acts as the basis for other more complex features and is not as useful by itself. However, these automatically added callbacks can be used to eliminate the manual labor of adding similar tracking code.

Tracing in General. This first improvement makes it possible to track the execution of Java methods and to write the information to the standard system output or a log file. This kind of tracking of code execution by log messages is generally known as tracing. The trace messages are usually disabled during normal development and in final builds to prevent excessive amount of log messages. Then, if more information is needed by the developer, the trace level messages can be enabled.

Automated Tracing. Using the implemented tool, the developer can automatically get trace messages without the need to add source code individually to each traced method. All that the developer needs to do, is to enable the method traces option in the instrumentation phase. No modifications to the existing code are needed. This option only outputs the traces in a simple predefined format to standard system output. The developer can implement specific callback methods himself if more control over the output is wanted. No references to any external Java libraries are needed, since callbacks to the method will be generated in the instrumentation phase and there is, for example, no need to register a listener. The location and name of the callback method name can be defined in the instrumentation phase. This callback can then be used to print information about each method invocation in any way wanted. A separate callback is available for both entering a method and exiting a method.

Modifying the Bytecode. Adding code to the beginning of a method is a straightforward operation. On the other hand determining when a method exits is a bit more complicated because there can be multiple return statements and thus multiple possible exit points. In addition, an exception can cause the method to exit at any point. The method tracing only generates the exit callback if the method exits normally. The more complicated special cases caused by exceptions are ignored for now. The improved stack traces explained in the following can be used to detect the thrown exceptions. When exceptions are ignored, all that is needed to detect exiting from a method is to find all the return statements in the method and to add additional bytecode for handling the callbacks just before them. The end of a void method is also considered as a normal return statement in bytecode. The instrumentation phase can be demonstrated with an example Java method. Listing 1 defines a method that calculates the mean for an array of integers. From the instrumentation point of view it is irrelevant what the method actually does, but it was chosen because it is short and meaningful but it still has enough code to demonstrate some special cases.

Listing 1. Java method before it has been instrumented.

```
public int calculateMean(int[] values) {
    if (values == null) {return 0;}
    int sum = 0;
    for (int i = 0; i < values.length; i++) {sum += values[i];}
    int mean = sum / values.length;
    return mean;
}
```

The Java source code in Listing 2. demonstrates how instrumenting the class with method tracing enabled affects the class. A call to a static method *Trace.methodEntered* has been added to the beginning of the method. This is a call to the method that handles the tracing for entering a method. This method will invoke a callback method that has been defined in the instrumentation phase. The *Trace* class is a Java class that is a part of an implemented run-time library. This and some other utility classes need to be included to the final application JAR package for the tracing to work. Also, a call to *Trace.methodExited* has been added before each return statement. This call will relay the information about exiting the method to the user defined callback method. The actual insertion is done as bytecode but if the resulting class would be decompiled it would result in source code that would be similar.

Listing 2. Same method after instrumentation.

```
public int calculateMean(int[] values) {
    Trace.methodEntered("MyMath", "calculateMean", 1);
    if (values == null) {
        Trace.methodExited("MyMath", "calculateMean", 2);
        return 0;
    }
    int sum = 0;
    for (int i = 0; i < values.length; i++) {sum += values[i];}
    int mean = sum / values.length;
    Trace.methodExited("MyMath", "calculateMean", 6);
    return mean;
}
```

More detailed information regarding the method that is being executed is provided in the callback method parameters, including the name of the class and the name of the method. The last parameter is the source code line number. It is added to make it possible to later distinguish multiple methods with same name, and to help finding the method in a source file. The class name and the method name are known in the instrumentation phase and the line number is also available if debug information has been included in the class file.

3.2 Improved Stack Traces

Stack trace is a structure representing the call path of a thread at some specific time of execution. It lists the method invocations that lead to the state that the trace represents. The position in source code where each method was invoked can also be stored to the structure. Stack traces are very useful when trying to locate and fix errors in the source code. The trace can be displayed when an exception has happened to easily find the exact place in source code what caused the exception. Unfortunately getting stack traces with full source line number information can be a problem when using CLDC.

Tracking the Execution of an Java Application. It is not possible to access the stack trace information programmatically in CLDC, but it is possible to solve the problem of not being able to get the stack traces with another approach. The solution is to modify the bytecode of the application itself, so that the application keeps track of its own execution. Each method needs to be modified to detect when the execution of a thread enters the method and when it leaves the method. In addition, each thread needs to be associated with a data structure that defines the current method call path of that thread. As the application itself will be keeping track of the execution of its code, it is possible to get the current stack trace at any point regardless of the execution environment.

The structure used to define the invocation path is a stack, where each element represents a method invocation that was done from the previous element. This stack mimics the actual call stack that keeps track of the method execution in the virtual machine [6]. Similar kind of structure has been available for the programmer in Java SE version 1.4 and in newer versions, where it is possible to call *throwable.getStackTrace()* method to receive an array of *StackTraceElement* objects. Java version 1.5 (or 5.0) takes this even further and provides multiple method in the *Thread* class for accessing the current stack trace programmatically. As CLDC is based on Java 1.3, neither *getStackTrace()* method nor *StackTraceElement* class are available [2,3,7].

An element representing the current method is pushed to the stack each time a thread enters a method and popped when the method is exited. It is also necessary to catch possible exceptions and pop the method if one is detected to keep the stack correctly up-to-date. Each time a method is entered and the thread is not known from before, a new stack trace is created and associated with that thread. Similarly, as the last item is popped from a stack, we know that the stack trace related to that thread is not needed anymore.

At the same time it is also possible to solve the problem of not getting source code lines in stack traces. This is however only possible if the required debug information has been enabled when compiling the classes that are being instrumented. If this line numbers debug information is enabled, the compiler adds a special bytecode instruction defining the source line number before each group of bytecode instructions that were generated from the same source code line. When instrumenting a method every time this kind of bytecode instruction is encountered instructions need to be added to update the current stack trace. The added instructions need to update the top element in the stack of method elements to point to the correct source line.

Adding Bytecode to Track the Execution. The instrumentation phase is demonstrated with the same example Java method that was used previously in Listing 1. The Java source code in Listing 3 demonstrates the effect of instrumenting with improved stack traces enabled. The *Trace* and *StackTrace* classes are normal Java classes that are also part of the implemented run-time library. Most of the functionality related to the stack traces is implemented as normal Java code in the run-time library to make the instrumentation simple. Amongst other things, the included utility classes keep track of the threads and map them to the right stack traces.

Listing 3. The same method after instrumentation with stack tracing enabled.

```
public int calculateMean(int[] values) {
    StackTrace trace = Trace.push("MyMath", "calculateMean", 1);
    try {
        trace.setSourceLine(2);
        if (values == null) {
            trace.pop();
            return 0;
        }
        trace.setSourceLine(3);
        int sum = 0;
        trace.setSourceLine(4);
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        trace.setSourceLine(5);
        int mean = sum / values.length;
        trace.pop();
        return mean;
    } catch (Throwable t) {
        trace.exception(t);
        throw t;
    }
}
```

A call to *Trace.push()* is added to the beginning of the method. This method pushes a new element, representing this method, to the call stack of the current thread. The name of the class, the name of the method, and the method beginning line number are passed as parameters. The parameters are needed later to print a stack trace that looks like a normal Java stack traces. For convenience the current stack trace element is saved to a local variable called *trace*.

The *trace.pop()* method pops the top element from the stack. A call to it needs to be added before each return statement to keep the stack state up-to-date. This is done similarly as with the method tracing before.

The whole original method is enclosed inside a try-catch block to catch any exceptions that would otherwise escape. Any exception that escapes the original method will be caught and thrown again, so that the functionality of the method does not change. Before throwing the exception, *trace.exception()* is invoked to update the stack trace. Calling *trace.exception()* pops this method from the stack and records the thrown exception. In normal Java source code it is actually not possible to throw a *Throwable* object without declaring it to be thrown in the method declaration, but it is possible on the bytecode level.

A call to *trace.setSourceLine()* needs to be added before each original line of code, so that the line number that is being executed is updated to the stack trace. If an exception will be thrown when executing the following original line, the trace will now point to the same line. In the example code, the execution can jump back when the for loop is being executed and the source line will not be set correctly. In the actual bytecode implementation this is not a problem as the call to the *trace.setSourceLine()* method is placed just after the bytecode instruction defining the line number debug information so that it is set correctly regardless of any jump instructions.

For the tracking to work, each thread needs to have its own stack trace. Every time a *Trace.push()* is invoked we need to find out which thread invoked the method. The current thread is resolved inside the *Trace.push()* method using *Thread.currentThread()*. A separate mapping is needed to find the correct stack trace for the current thread as the thread object itself is part of the standard Java library and cannot be modified.

Logging Exceptions Traces. Now that the application itself keeps track of the execution of its threads, it is possible to get a up-to-date stack trace at any given point in the code. Similarly to Java SE, the tools provide the user with an API to receive the trace explicitly as an array of method elements.

In exception situations it is convenient to log the stack trace implicitly without any code written by the developer. The tool provides an build time option that can be used to determine how the exceptions are logged. The options are to log all exceptions, to log only uncaught exceptions, or to not log exceptions implicitly at all. All that the developer would need to do is to define a callback method that will be invoked each time an exception has occurred.

It is a common practice to use *throwable.printStackTrace()* to print the stack trace to standard output whenever an caught exception could be interesting when debugging. As stated before, the standard output is often not available and the stack trace should be redirected to the logging system that is in use. This can be achieved by detecting each call to *throwable.printStackTrace()* in the instrumentation phase and modifying each call so that the stack trace of the exception will also be logged.

Listing 4 provides an example method that will be used to demonstrate how the exception logging works. This method invokes another method called *con-*

nection.open() that is known to possibly throw an *IOException*. If an exception is thrown, it will be caught and the exception stack trace will be printed.

Listing 4. A Java method catching a possible exception and printing the stack trace of the exception to the standard output stream.

```
public void open() {
    try {
        connection.open();
    } catch {IOException ioe} {
        ioe.printStackTrace();
    }
}
```

Now we want the exception stack trace to appear in the logging system that is used by the application. As it is necessary to detect exceptions to keep the stack trace correctly up-to-date, it is easy to add a callback to some logging code each time an exception is detected. The stack trace can be given as a parameter. In addition, we need to detect exceptions that are caught by the original method. Otherwise it is not possible to detect exceptions that never leak outside the method they were thrown in. Exceptions that are caught inside the original method need different kind of processing as they should not pop the method from the call stack.

Listing 5 shows how the instrumentation modifies the bytecode of the method to enable detecting all exceptions. All the same modifications are done as before when adding the general stack tracing (Listing 3). In addition, code is added to detect caught exceptions and to enable their logging.

A call to *trace.exceptionCaught()* is added to be the first instruction in the catch block of the original code (after line 4). This call needs to be added to the beginning of every catch block in the instrumented bytecode. If the developer has chosen to print all exceptions, a callback to the logging method will be done each time *trace.exceptionCaught()* is invoked. Otherwise the current stack trace is just saved so that the callback can be called later with correct stack trace if necessary. The stack trace needs to be saved before any code is executed inside the catch block to be able to get the correct source line information later. Any possible finally blocks do not need changes as they do not catch exceptions.

Finally, a call to our own *Trace.printStacktrace()* needs to be added after every standard *throwable.printStackTrace()* method. The added method call logs the stack trace that was saved when the exception object that is given as a parameter was detected. It uses the callback method defined by the developer to log the trace.

If the developer has chosen to implicitly print all uncaught exceptions, all we need to do is to add an extra check inside the final *trace.exception()* method that is invoked when an exception escapes the original method. If the call stack

Listing 5. The method after instrumentation for logging exception traces.

```
public void open() {
    StackTrace trace = Trace.push("MyClass", "close", 1);
    try {
        try {
            trace.setSourceLine(3);
            connection.open();
        } catch {IOException ioe} {
            trace.exceptionCaught(t);
            trace.setSourceLine(5);
            ioe.printStackTrace();
            Trace.printStacktrace(throwable);
        }
    } catch (Throwable t) {
        trace.exception(t);
        throw t;
    }
}
```

is empty after popping the current method we know that the exception was not caught by any instrumented code and it should be logged. An extra warning message is shown when a thrown exception escapes uncaught.

3.3 Deadlock Detection

Synchronization is required to keep data from corrupting when multiple threads are using shared resources. Synchronization restricts the access to certain memory areas so that only one thread may access the data at a time. In Java, synchronization is implemented on the language level and also the Java virtual machine has its own specialized instructions for it [4,6].

Adding synchronization adds the possibility of deadlocking however. A deadlock happens when multiple threads are accessing synchronized resources so that a thread has to wait for another thread that is directly or indirectly waiting for the original thread.

Errors that cause deadlocks are usually hard to detect and hard to debug. The deadlock may only happen in some special case with certain specific timing. There are many techniques that can be applied to prevent deadlocks from happening and it is usually best to apply some well known strategy to rule out the possibility of a deadlock. One commonly applied strategy is called *resource ordering*. When using resource ordering deadlocks can be avoided by accessing the resources always in the same order.

Even if preventing deadlocks has been taken into account when developing an application, it is still possible to miss a possible deadlock situation. If a deadlock is already occurring and it can be reproduced, a good way to find the cause for the deadlock is trying to detect it on run-time and print the stack trace for each of the deadlocking threads. Java SE provides a built-in method for finding deadlocked threads when the application is running. The *findMonitorDeadlockedThreads()* method in *java.lang.management.ThreadMXBean* class can be used to find threads that are currently deadlocked [3]. The same functionality is not available in Java ME, but similar functionality can be implemented by instrumenting the applications bytecode.

Bytecode Manipulation to Detect Deadlocks. The application needs to be modified to track when each of the running threads enters a synchronized block and thus acquires a lock. Every synchronized method and synchronized block must be modified to be able to detect deadlocks when they happen on run-time. Before entering a synchronized code block the current thread is marked to be waiting for the lock object defined in the synchronization block. Just after entering the synchronized area the thread needs to marked as the one owning the lock. Similarly the lock is marked as released when the method or block is exited.

A data structure containing the locks acquired by each thread needs to be maintained. Each time a new lock is requested by a thread, the data structure must be checked to see if a deadlock situation has occured. When a deadlock is detected the stack trace of all the threads that are causing the deadlock can be printed using the same method as earlier when printing normal exception stack traces.

For synchronized member methods the lock object is the *this* reference and for static methods it is the *Class* object for the class in question [4]. Using the synchronized keyword in a method declaration has the same effect as enclosing the whole method in a synchronized block. As, an example, let us consider the following Java method:

```
public synchronized void deadlockTest() {connection.open();}
```

When a synchronized method is instrumented, the synchronized keyword is removed, and a separate synchronized block is added to achieve the same effect. This needs to be done because we need to mark the thread as waiting for a lock before it enters the synchronized block. Listing 6 shows the instrumented version of the method from Listing 6. The resulting code for Listing 6 would be exactly the same, but the line number have been aligned to match the former example. The added instructions for tracking stack traces have been excluded to make the example clearer. In addition, the current stack trace must be passed to the deadlock detector so that the current thread can be determined and the stack trace printed upon a deadlock.

The call to *DeadLockDetector.waitingForLock()* method is added to mark the thread as waiting for a lock. Waiting is not necessary if the thread already owns the lock. A call to *DeadLockDetector.locked()* is added right after entering a synchronized block to mark the current thread as the owner of the lock. A counter must be increased to be able to mark the lock as released when the

Listing 6. The method after it has been instrumented for deadlock detection.

```
public void deadlockTest() {
    DeadLockDetector.waitingForLock(this);
    synchronized (this) {
        DeadLockDetector.locked(this);
        connection.open();
        DeadLockDetector.released(this);
    }
}
```

outermost synchronized block using the same lock is exited. Before the end of the block a call to *DeadLockDetector.released()* method is added. This call decreases the counter for the used lock and marks the lock as released if this was the outermost block using this lock. The lock object reference that is used for synchronization is given to all the method calls as a parameter

The code shown in the example is not enough to correctly handle exception situations. Fortunately in Java bytecode the instruction for releasing a locks is automatically added to each exception catch block when a class is compiled. The *DeadLockDetector.released()* method call just needs to be added before each instruction releasing a lock. The case when a method is exited with an exception must be however handled separately and all the acquired locks must be released. The exception tracking described earlier can to be used for that.

The *DeadLockDetector* class handles the logic for detecting deadlock situations. The class keeps track of the locks that are owned by each thread and each time before entering a synchronized block a check is made if a deadlock situation has occurred. A deadlock occurs if the lock that the thread is waiting for is locked by another thread that is again waiting for the current thread. Each time when trying to acquire a lock a check needs to be made for these kind of cyclic locking situations.

In addition to synchronized blocks and methods, also *object.wait()* methods affect the ownership of locks and can cause deadlocks. A wait method invocation stops the thread and releases the lock until a notification wakes up the thread again or a timeout happens. When the thread is woken up it tries to acquire the lock again and can cause a deadlock.

Listing 7 shows how each call to a *object.wait()* method in the original class is modified. before the call a call to *DeadLockDetector.wait()* is added. This method call marks the lock as released and marks this thread as waiting for the lock again. Even though this thread might sleep for a long time and is not technically waiting for the lock, it is close enough for the purpose of deadlock detection, as the next executed instruction cannot be reached before the lock is acquired. After the wait, a call to *DeadLockDetector.locked()* is added as the lock is now again acquired by this thread.

Listing 7. Instrumenting wait calls to enable deadlock detection.

```
...
    DeadLockDetector.wait(this);
    this.wait();
    DeadLockDetector.locked(this);
...
```

3.4 Simple Profiling

The performance of an application may be found to be be less than satisfactory at some point of development. The process of determining which parts of the application are using the most resources is called *profiling*. Optimization in these so called hot spots will also give the most benefit. If optimizations are done without knowing what the actual performance bottlenecks are, a lot of time can be wasted without much improvement in performance. Profiling can be used to find problems in memory use and to find execution speed bottlenecks [11].

Bytecode Manipulation to Enable Profiling. The basic idea of the implemented profiling tool is to provide information about the time that is spend executing in each of the methods of the application. This can be achieved by getting the system clock time before executing a method and the time after the execution. The difference in the time can be used as an estimate of the amount of time spent in the method. Also the number of calls to each method is stored. The memory usage of the application is not analyzed by the implemented tool. Let us next consider the following Java method that is to be instrumented with profiling information:

```
public void profilingTest() {connection.open();}
```

Listing 8 shows the example method after instrumentation. A line is added to the beginning of the method that stores the starting time of the execution of the method to a local variable. Code for determining how long the execution took is added to the end of the method. The difference between the start time and end time is calculated and added to the total time used in the method. After that the counter for the number of method calls is incremented.

An array is used to store the profiling information. Reference to the array is kept in a static field called *$methods*. This field needs to be added to each instrumented class. The array contains *ProfiledMethod* objects. Each object holds the profiling data for one specific method: the total time spent in the method and the call count for the method. Each method of a class is associated with an index number, which is used to access the correct object in the array.

Listing 8. The same method after it has been instrumented for profiling.

```
public void profilingTest() {
    long start = System.currentTimeMillis();
    connection.open();
    $methods[0].totalTime +=
        System.currentTimeMillis() - start;
    $methods[0].callCount++;
}

static final ProfiledMethod[] $methods = new ProfiledMethod[1];

static {
    $methods[0] = new ProfiledMethod("profilingTest()");
    Profiler.addClass("MyClass", $methods);
}
```

The profiling information array needs to be initialized in the static initialization block so that every method contained in the class is added to the array. Also the name and signature of the method is saved so that the data for different methods can be identified later. The array is also registered to the *Profiler* class with the name of the current class. The profiling information can be later accesses using methods that are available in the *Profiler* class.

4 Evaluation

Using Java ME and SE simultaneously made the old and limited CLDC libraries feel even more limited compared to the less restricted Java SE. The full Java library of Java SE and the new Java language features will still be missed after the implemented improvements.

4.1 Limitations and Potential Problems

All the standard Java libraries and other libraries that are part of the platform cannot be instrumented. Fortunately, the interesting parts of the code that is being debugged are usually the ones that are being developed and therefore it is possible to instrument them. For performance reasons it might make sense to instrument only part of the whole project.

Tracing is not possible in classes that have not been instrumented. For example if an instrumented method calls another uninstrumented method that again eventually calls instrumented code, we have no way of knowing what happened in the uninstrumented method and if there were other method calls before calling the instrumented code again. In these situations the stack trace will just contain an unknown element to mark the uninstrumented code. Similar problems exist with deadlock detection where some deadlocks may not be detected if all of the code is not instrumented.

To include correct line numbers in stack trace, the instrumented class files are compiled with debug information. If the application uses some third party library whose source code is not available, some parts of the application might not include the necessary debug information. Source file name and line number information is needed to map the byte code instructions in the compiled classes to source code lines. Stack traces will work even if the information is missing but the stack traces will not contain the line number and source file name.

When the optimizations are enabled in the instrumentation phase, the instructions for catching exceptions are not added to some very simple methods. This has the side effect that some exceptions that can happen at any point of execution may be reported to have happened on a wrong line. Normally this is not a problem and the optimizations can be disabled if necessary.

Instrumenting modifies the Java bytecode and can also cause issues with code that was previously working. Many errors are detected immediately when the preverification fails but some errors may only occur on run-time. Extensive testing has been done to make sure as many as possible different kind of Java language constructs work without problems.

Biggest problem with the implemented deadlock detection is that the instrumentation affects the timing of the code execution. As the deadlock may only happen on some specific timing it is possible that the deadlock cannot be reproduced when the detection is enabled. Similar problem also exists in profiling: The profiling results are affected by the code that is added to measure performance.

4.2 Performance and Size Impact

The instrumentation phase adds bytecode instructions to every instrumented method. The added instructions make each class bigger and affect the execution speed of the program. The added instructions also include calls to the runtime library part of the debugging tool. These runtime library classes need to be included in the final JAR package and require about 10 kilobytes of space. Also the memory usage of the application is slightly increased as each thrown Exception and the current stack trace for each thread needs to be kept in memory when running an instrumented MIDlet. The overall size and speed overhead of the instrumentation was tested using some existing Java ME benchmarking software and using an actual Java ME application. The memory usage impact of the implemented tools was not measured as the increase in memory usage is very small compared to the amount of memory that is usually available.

JBenchmark (*http://www.jbenchmark.com/*) is a set of Java ME benchmarking tools that are meant to measure the performance of Java ME enabled devices. The differences between the performace of different devices is not interesting from point of view of this paper. However, the same benchmarking MIDlets can be used to measure how the performance changes after instrumenting the benchmarking MIDlet to enable the improved debugging features. Multiple versions of the JBencmark MIDlet were used to measure both the impact on the MIDlet size and the impact on the MIDlet execution speed.

Table 1. The size impact of instrumentation

Test MIDlet	**Original size**	**Size after instrumentation**	**Size increase**
JBenchmark	26236 bytes	28747 bytes	9.6%
JBenchmark 2	63994 bytes	67794 bytes	5.9%
JBenchmark Pro	207103 bytes	238981 bytes	15.4%
Dromo client	513167 bytes	671686 bytes	30.9%

When instrumenting, the size of the original uninstrumented classes is increased approximately 5–30%, depending on the application and the instrumentation parameters. Table 1 shows the size impact of instrumentation on some existing Java ME applications. The constant increase of approximately 10 kilobytes caused by the runtime library classes is ignored in these calculations so that it will not skew the size increase percentage. The overhead is smaller in the JBenchmark MIDlets as they do not include debug information and thus the information to track line numbers in exception will not be added to them. JBenchmark and JBenchmark 2 were also already obfuscated before instrumentation was done. This reduces the size increase as the string literals that need to be added when instrumenting are much shorter. The *Dromo client* (*http://sourceforge.net/projects/dromo/*) is an open source MIDlet for remotely controlling a video recording server. It was chosen to represent a real life Java ME application. The used version was compiled to include all the necessary debug information and it was not obfuscated.

The maximum allowed size of a MIDlet JAR package can be very restricted depending on the target device. The limit can be as little as 100 kilobytes in some older mobile devices. In current devices the limit is considerably higher, up to megabytes. A 30% increase in MIDlet size can become an issue in some cases. The MIDlet can be obfuscated after the instrumentation to reduce the size if necessary. Obfuscation renames all the classes and normally this would also make the stack traces unreadable. If the instrumentation is done before obfuscation, the original class and method names will be preserved even after obfuscation in the stack traces. This is possible because the method and class names are literals in the code and will not be changed by the obfuscator. The reduction in JAR size is not as much as it would be without instrumenting. Obfuscation also reduces the size of the runtime classes that are added to the JAR by the implemented tools.

Each method call is instrumented with multiple new calls to methods in the runtime debugging classes and the current source line number needs to be updated on each new line of code. If an instrumented method is called repeatedly in a loop, this can add up to a significant amount of time. The speed impact of instrumentation is shown in Table 2. Each benchmark was first run uninstrumented and then instrumented. The shown percentage shows the performance of the instrumented benchmark relative to the uninstrumented run. All the measurements were done using the default emulator of Sun's Wireless Toolkit and a Nokia X3 device. The table shows an average of three runs of each benchmark.

Table 2. The speed impact of instrumentation

Benchmark	**Speed after instrumentation**	
	WTK emulator	**Nokia X3**
JBenchmark	97.3%	96.3%
JBenchmark 2	84.9%	97.5%
JB Pro: Business math	89.2%	99.8%
JB Pro: Chess AI	23.1%	9.4%
JB Pro: Game physics	28.8%	20.6%
JB Pro: Image processing	22.9%	17.1%
JB Pro: Shortest route	46.6%	15.6%
JB Pro: XML parsing	85.7%	87.8%
JB Pro: ZIP compression	12.9%	7.6%

The speed impact varies a lot depending on the code that is being instrumented. The JBenchmark and JBenchmark 2 tests mostly consist of different kind of graphics operations where the majority of execution time is not spent in the Java code and the impact on MIDlet performance is very small. JBechmark Pro was used to run some specific processing intense operations. These operations are closer to the worst case scenario where a very simple method is run a huge number of times in a loop. Also the performance impact is very significant in these kind of tests.

In UI and graphics operations the application speed is not affected very much and even a slowdown to half or one fourth of the original speed is normally acceptable when debugging an application. Instrumentation can however cause the application to slow down up to one tenth of the original speed in some processing intense operations.

The performance degradation and the increase in program size can further be tackled by different options in the instrumenting phase. It is possible to limit the number of instrumented classes by including only interesting classes, or by excluding some classes that are uninteresting or processing intense. Another option is to disable some tracing features. Both of the options compromise the amount of information that will be available when executing the program.

5 Conclusions

This paper presents a set of tools to make Java ME development and debugging faster and easier. Implementing dynamic tools that are used when the application is running is challenging in the very restricted sandbox of Java ME. As a workaround, we instrument the compiled application binary statically before running it. This way the application can itself gather information at run-time.

The developed tools have already been used in Java ME projects being developed at Sasken Finland. The feedback from the developers has been very positive. Mainly the ability to get stack traces has been used, and it has proven its usefulness multiple times already during a short period of two months.

References

1. Java^{TM}2 Platform Standard Edition 1.3 API specification, `http://java.sun.com/j2se/1.3/docs/api/` (accessed on February 2010)
2. Java^{TM}2 Platform Standard Edition 1.4 API specification, `http://java.sun.com/j2se/1.4.2/docs/api/` (accessed on February 2010)
3. Java^{TM}2 Platform Standard Edition 5.0 API specification, `http://java.sun.com/j2se/1.5.0/docs/api/` (accessed on February 2010)
4. Arnold, K., Gosling, J., Holmes, D.: The Java^{TM}Programming Language, 4th edn. Addison-Wesley (April 2008)
5. Lau, A.: The fragmentation effect. JavaWorld (May 2004), `http://www.javaworld.com/javaworld/jw-05-2004/jw-0524-fragment.html`
6. Lindholm, T., Yellin, F.: The Java^{TM}Virtual Machine Specification, 2nd edn. Prentice-Hall (April 1999), `http://java.sun.com/docs/books/jvms/`
7. Sun Microsystems, Inc. Connected Limited Device Configuration (CLDC) Specification 1.1 (March 2003), `http://jcp.org/aboutJava/communityprocess/final/jsr139/`
8. Sun Microsystems, Inc. Mobile Information Device Profile (MIDP) Specification 2.1 (June 2006), `http://jcp.org/aboutJava/communityprocess/mrel/jsr118/`
9. Tanter, É., Ségura-Devillechaise, M., Noyé, J., Piquer, J.: Altering Java Semantics via Bytecode Manipulation. In: Batory, D., Blum, A., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 283–298. Springer, Heidelberg (2002)
10. Topley, K.: J2ME in a Nutshell. O'Reilly (March 2002)
11. Wilson, S., Kesselman, J.: Java^{TM}Platform Performance: Strategies and Tactics. Addison-Wesley (June 2000)