TOOLS FOR INTEGRATING FORMAL METHODS INTO THE JAVA SOFTWARE DEVELOPMENT PROCESS (AN EXTENDED ABSTRACT)

Sriram Sankar

Metamata, Inc.
sriram.sankar@metamata.com

Abstract: We study different techniques by which formal methods can be applied within real software development environments for Java. We use the Metamata Toolsuite (our software development environment for Java) to experiment with these techniques. The goal is to improve the value of these environments from the point of view of a customer. Essentially, this means that the formal methods considered must be made extremely practical, while providing significant value for the average Java developer. We describe various kinds of formal methods we have defined and experimented with so far. One of our goals is to create an interest in formal methods enthusiasts to develop more such practical formal methods that can be made available to Java developers not conversant with this field.

INTRODUCTION

Practical formal methods have always been of interest to me ever since the early 1980's. My career has taken me from a Ph.D. student in formal methods to academic research from which I moved on to a corporate research laboratory. From there I moved to a development team, and finally to a software tools start-up company. My focus has therefore continuously shifted to being more and more practical and customer oriented. While we do a lot of work nowadays that cannot be categorized as formal methods, we have spent a lot of time thinking about how we can leverage various aspects of formal methods to provide an important added value to the tools we build. Clearly, our focus has to be on extremely practical formal methods that the average programmer can avail of and obtain reasonable gains in software quality.

In this paper, we discuss various different kinds of formal methods that we have developed and incorporated into our toolsuite. We encourage our reader to extend our ideas even further in the same direction (*i.e.*, towards practical use by the average developer). The pur-

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

pose of this paper is to demonstrate that it is possible to make formal methods usable by a large group of developers for gains in software quality.

The next section provides an overview of the Metamata Toolsuite, a software development environment for Java. Each of the following sections then go on to discuss different ways in which we offer formal methods of different kinds. First we discuss traditional assertion capabilities and tool support for it. The following section describes constraint specification for thread synchronization constructs. Then we discuss support for source code editing to prevent certain kinds of mistakes. Finally we talk about object-oriented metrics as a way of measuring software quality and then conclude the paper.

This paper clearly builds upon a lot of earlier work on formal methods. The original contribution here is not in the formal methods themselves, but in the understanding of what it takes to make them practical and appealing to the software developer.

THE METAMATA TOOLSUITE

The Metamata Toolsuite is a software development environment that provides capabilities for static and dynamic analysis of Java programs. In addition it also offers editing capabilities through which Java code and assertions can be written. The toolsuite currently offers two kinds of static analysis tools — Audit, a tool that checks Java code for errors of style, performance, maintenance, *etc.*, and Metrics, a tool that applies standard formulae of object oriented QA metrics on Java programs. The toolsuite also offers a debugger.

All tools have been built in an extremely flexible manner to facilitate easy plugging in of additional capabilities such as described in the following sections. Furthermore the tools integrate very well with each other — something that is important to facilitate more easy acceptance of new features.

More details of the Metamata Toolsuite may be obtained at http://www.metamata.com.

THE DIAGNOSTICS CAPABILITY

A formal specification is a construct that says "what" a program does and not "how" it does it. In some sense, a formal specification is like a comment on the program code. One of the most useful practical application of formal specifications is to execute it to determine if the program is indeed meeting its specifications during that particular run. For this purpose, we have provided two capabilities for specification of Java programs:

- *The Diagnostics API.* This is a set of methods defined in a class called `Diagnostics`. This class provides methods for stating assertions, for stating that certain portions of code is unreachable, and also for stating that certain portions of code is incomplete. In addition, this class provides methods that simply print strings to output. An extremely condensed form of this API is shown at the end of this section. The complete API is available for download from http://www.metamata.com as part of the toolsuite.

- *Diagnostic code.* A means of identifying Java statements as not part of the primary program, but rather part of the diagnostic code meant to check properties about the program. This is achieved by labeling such statements with the label `DIAG`.

We believe users can only cope with a very straightforward set of specification capabilities. The syntax of the specification capabilities must be as close to the underlying programming language as possible to make it easy to use. Furthermore, such specification constructs need strong language support to make them useful and to encourage developers to use them. The Metamata Toolsuite offers two support capabilities for the diagnostic features:

- *Debugger support.* The Metamata debugger understand specification features and handles them appropriately. For example, it provides breakpoints on assertion failure, or on reaching unreachable code, or incomplete code.

- *Packager support.* Perhaps the most important tool support (without which users will not use the diagnostics features) is the ability to automatically remove the diagnostics constructs from the user program.

The condensed form of the Diagnostics API follows:

```
package com.metamata;

public final class Diagnostics {

    /**
     * Evaluates the condition and takes appropriate action if it
     * evaluates to false.
     */
    public static boolean assert(boolean condition) { ... }

    /**
     * Specifies that this portion of code should never be
     * reached. If it is, the appropriate action is taken.
     */
    public static boolean unreachable() { ... }

    /**
     * Identifies this area as containing incomplete code.  If
     * this statement is executed, appropriate action is taken.
     */
    public static void TBD(String message) { ... }

    /**
     * Used to print diagnostic statements during testing.
     */
    public static boolean println(Object value) { ... }

}
```

SPECIFYING CONSTRAINTS ON THREADS

Given the first-class status of threads in Java as well as the difficulty in writing robust multi-threaded applications, we cover constraints for threads separately in this section.

In our experience with Java development, we found the thread synchronization constructs provided by the Java language somewhat low-level (for the purpose of writing constraints, and consequently their direct use in programs). We therefore defined a `Lock` API that is built on top of the primitives Java offers.

The basic `Lock` API is shown below:

```
public interface Lock {

    /**
     * Acquire a lock on this object.  Blocks until the lock
     * is available.
     */
    public void acquireLock();

    /**
     * Relinquish a lock on this object.
     */
    public void relinquishLock();

}
```

The Lock API is a way for multiple threads to cooperate with each other through acquiring and relinquishing resources (locks). By doing this, we can implement critical regions, safely access shared variables, *etc.*

There can be different implementations of the Lock API that differ in detail (such as a single lock, multiple readers and single writers, *etc.*). But by providing a nearly identical API to clients, we can develop a general constraint specification capability.

With each lock object, we identify three different kinds of events:

- `requestLock`
- `obtainedLock`
- `relinquishLock`

Occurrences of these events may be logged to a file for postmortem analysis. Objects may also register themselves with these locks as listeners — they therefore get notified when-

ever these events occur. These listeners can then check constraints on these events and take action if any of the constraints are violated.

A straightforward constraint that can be checked either postmortem or through listeners is:

Lock A may not be requested while holding Lock B

This constraint is a typical deadlock avoidance scheme — where locks are ordered and if a thread wishes to acquire multiple locks, it must acquire them in a particular order.

A sophisticated debugger can also be made to listen to such events and check for invariants. The typical debugger command corresponding to the above invariant may be:

```
stop if A.requestLock() when B.holdingLock()
```

The debugger would then provide a breakpoint whenever a request for Lock A is made while holding on to Lock B. The user can then debug their program in the standard manner.

Providing this capability through the debugger makes this formal method feature much more appealing to the average user as compared to requiring the user to write constraints in some specification language.

EDITING SUPPORT AND STATIC ANALYSIS

When the user edits a Java file, the environment can attempt to offer support in ensuring that certain invariants are maintained. We have just started working on this area, hence only the few ideas listed below. We hope to enrich this list over time.

The most important category of support is in ensuring a consistent edit across the entire system. For example, when the user edits a method, the editor must automatically ask the user to ensure that other methods that override the edited method are also up-to-date. A method is typically edited because there is a bug that needs to be fixed, or because of a change in the features offered by the method. In either case, we are changing the behavior of the method to make it more in tune with the current (usually unwritten) specification of the method. Hence the user must ensure similar edits are performed on all methods that override this method — given that methods in subclasses usually obey the specifications of the method they override.

Another example of a check performed during editing is to ensure command query separation. It is good programming style to change state only in methods that do not return a value. Methods that return value should simply be observers on the state of the object and not change it. Static analysis can be performed during editing to warn the user of possible violations of the command query separation rule.

In general, it is good to perform static analysis to check for general program consistency after editing the program. In addition to the more interesting tests described above, it is also useful to perform an array of simple consistency checks also. For example, if a type inheritance hierarchy is modified, it is possible that type casts that exist in other parts of the user code are no longer necessary. This is quite easy to test.

Another useful editor feature is the automatic insertion of "`Diagnostics.TBD()`" into newly created blocks — this way, the user is automatically made aware of incomplete portions of their code.

COMPLEXITY METRICS

The Metamata Toolsuite has a component that evaluates quality and complexity metrics on Java source code. These metrics are formulae that may be applied on the Java program to obtain numbers which correlate in different ways to the quality or complexity of the Java program.

The area of Metrics has been studied for a long time and the quality assurance community has come up with many different formulae to measure different aspects of quality and complexity. In many senses, these formulae are heuristics — *i.e.*, their result is a good guess on quality and complexity, but not an absolute statement.

We believe that a similar approach to building formal methods metrics may be beneficial. We encourage formal methods researchers to explore ways of approximating compute intensive formal methods into much simpler formulae which may be applied as formal methods metrics on programs. While these formal methods metrics may not provide as precise information as a full fledged proof or analysis, it can still provide rough, but useful information rather quickly.

CONCLUSIONS AND FUTURE PLANS

This paper describes work in progress. Clearly, there is a lot of possibilities for extensions to these techniques to introduce formal methods into real development environments in a non-intrusive manner. We are just beginning to collect feedback from users on the benefits of applying very simple formal methods with strong tool support. We hope to see a lot of positive feedback and also hope to extend our formal methods support further.

One specific area of future work is in the area of multi-process testing and debugging. The Metamata debugger provides an ideal infrastructure in which to perform this activity. Our goal is to add some simple specification capabilities to this tool to aid the user in their multi-process testing and debugging. The constraints on the Lock objects is an obvious candidate to extend from a multi-threaded environment to a multi-process environment.

We encourage readers to provide feedback to improve on the ideas described in this paper, and also to contribute new ideas.