MVT user's manual

Jan Lönnberg
jlonnber@cs.hut.fi
http://www.cs.hut.fi/~jlonnber/

14th July 2005

1 Introduction

As software has grown more complex, the amount of errors in it, known as *bugs*, has increased. Market pressures can further compound this problem by causing a project to be developed with unskilled programmers or insufficient time or money. It is estimated that software errors lead to costs of tens of milliards of euros every year.

Bugs are essentially a difference between the intended behaviour of the program and its actual behaviour. Thus, one way to find and eliminate bugs (an activity known as *debugging*) is to examine the operation of the program and compare this to the desired operation. This approach is called *testing*. Tools that assist in debugging by allowing programmers to examine the current state of a program (which includes the data the program is working with in memory and the currently executing code) and control its execution are called *debuggers*.

Current debuggers have several limitations. As they generally show data by displaying the values of individual variables, it is often hard to see the interesting aspects of the running program and its data. Object-oriented development has allowed programmers to hide unnecessary detail while developing, but debuggers generally do not take advantage of this. Furthermore, it is difficult to test results of operations in the program without writing additional code that runs parts of the program and examines the results. Also, when a problem is found, its cause is often lost in the past, which necessitates careful rerunning and stepping through the program to find the cause of the problem.

The goal of *MVT* (*Matrix Visual Tester*) is to simplify testing and debugging. It does this by providing useful visualisations of executing programs and their data (both the current and previous states) and a graphical interface that allows the user to modify data values and invoke methods at will, in addition to the facilities traditionally provided by debuggers such as stepping and breakpoints.

2 Installation

MVT requires some software to compile and to run. This is shown in Table 1. These programs and libraries should be installed before attempting to compile or run MVT.

Unpacking MVT will create a directory called `MVT`. Similarly, the other programs and libraries mentioned above will be installed in directories of their own, as shown in Table 2. In this document, I will use Ant naming conventions for directories and files; the user's home directory is therefore `${user.home}` and the directory separator is a forward slash.

In order to compile MVT, you must edit the property definitions for `${bcel}` and `${matrix}` in the build file `${mvt}/build.xml` to reflect the actual installation location of BCEL and Matrix, unless you install these programs into the default locations specified in Table 2.

Name	Available from	Compile	Run
Sun Java SDK v. 1.4.2:	http://java.sun.com/j2se/1.4.2/download.html	×	×
Jakarta BCEL v. 5.1:	http://jakarta.apache.org/bcel/	×	×
Matrix v. 9:	http://www.cs.hut.fi/Research/Matrix/	×	×
Apache Ant v. 1.5:	http://ant.apache.org/	×	

Table 1: Software required to compile and run MVT

Name	Directory	Default location
Java SDK	${java.home}	—
BCEL	${bcel}	${user.home}/jakarta-bcel
Matrix	${matrix}	${user.home}/matrix
MVT	${mvt}	${user.home}/MVT

Table 2: Default installation locations

Assuming the current directory is `${mvt}` and Ant is in the command search path, the commands in Table 3 can be used to compile MVT and its Javadoc documentation:

In order to easily start MVT and its instrumenter, you should create scripts that pass the right options to the JVM and place these scripts in your command search path.

The script `MVT` should execute the following:

```
${java.home}/bin/java -Xmx512m -cp
${mvt}:${java.home}/lib/tools.jar:${bcel}/bin/bcel.jar:${matrix}/code/matrix.jar
fi.hut.cs.mvt.ui.MVT
```

The script `instrument` should execute the following:

```
${java.home}/bin/java -cp ${mvt}:${bcel}/bin/bcel.jar
fi.hut.cs.mvt.connection.instrumentation.Instrumenter $@
```

In the above scripts, the correct installation paths for Java, BCEL and Matrix should be substituted. If `java` is in the command search path, the directory can be omitted. Both of the above commands should be on a single line. Also, users of non-Unix operating systems may need to change the directory separator from a colon to another symbol (a semicolon under Windows, for example). The number after `-Xmx` should be something like half the amount of RAM on your workstation (in megabytes); this is the maximum amount of memory available to MVT.

The directory `${mvt}/scripts` contains versions of these scripts for Unix and Windows systems that use the default installation directories in Table 2 (under Windows, the home directory is assumed to be `c:\`), have the Java 1.4.2 binary directory in the command search path and have an environment variable called `$JAVA_HOME` pointing to the directory to which the Java 1.4.2 SDK was installed.

3 Preparing a program for use with MVT

In order to allow MVT to monitor the execution of a program, the program must be compiled with debugging information (`javac -g`). Also, calls to MVT must be added to the program's bytecode. This is called *instrumentation*. In order to instrument a Java class,

Command	Effect
ant	Compile MVT.
ant clean	Delete compiled class files and Javadoc documentation of MVT.
ant javadoc	Generate Javadoc documentation for MVT.

Table 3: Commands to compile MVT and documentation

execute the following command:

`instrument class` ⟨class name⟩ ⟨class path⟩

In order to instrument all the classes in a package (including subpackages), execute:

`instrument package` ⟨package name⟩ ⟨class path⟩

You should instrument all classes that execute code that you wish to trace the execution of and whose instances you want to examine in detail. You should not instrument any standard library code, as this may have undesirable effects on performance and/or stability. To avoid this, the instrumenter does not automatically include the standard library classes in its class path. Note that instrumented classes require MVT to be in the class path in order to execute properly. The instrumentation can not be removed from a class file; you should recompile the class or restore it from a backup copy instead.

4 Starting MVT

Enter the directory you wish to use as the current directory for the debuggee (the program you wish to examine with MVT) and run the script `MVT`. MVT settings are loaded from this directory, allowing you to use different sets of settings for different programs. In most cases, you will probably want to copy the default settings (`*.ini`) from `${mvt}/defaults` to your working directory. If no MVT settings exist in a directory, a minimal set of settings will be created instead.

The MVT main window should appear. Use the **Load** menu to modify class path and source path settings as necessary, before loading the debuggee VM using **Load/Start**. The standard library and all classes required by MVT are automatically added to the class path.

5 The MVT main window

Once the debuggee VM has been started, the MVT main window contains (from top to bottom):

- The menu bar. This contains menus that have global effects.
- The structure panel, which contains the data view (described in Subsection 5.1), the package tree (described in Subsection 5.2) and method invocations that have not yet been started (described in Subsection 6.3).
- The source view. This shows the currently executing line in the source code.

5.1 The data view

The most important view in MVT is the data view, which shows the *variables* (memory addresses that contain a value that can be read and written by a Java program) in the debuggee grouped together in *data containers*. The variables are *local variables*, *array elements* and *fields*. Local variables belong to *frames* (each of which belongs to an invocation of a method) on the *execution stack* of a *thread*. Array elements form *arrays* and fields belong to *classes* or *objects*, which are instances of classes. The data view in MVT is based on showing the data containers as tables and the variables they contain as elements in the tables.

Primitive values are simply shown as text. Object references are represented by nesting the referenced object inside the referring object or using an arrow from the referring variable to the object that is referred to.

The current execution position is shown in two ways. Each stack frame is labelled with the currently executing method and line number in the source code. Also, the current line

in the topmost frame in the running thread is shown highlighted in the source code view below the data view.

MVT logs the execution of the debuggee. The user can step backwards and forwards through the execution history using the animator controls above the data view and in the **Go** menu. The data and code views in the main MVT window show the state of the program at the specified time. The user can also easily rewind the view to the last modification to a variable. Stepping back and forward through the log does not affect the actual debuggee. The **Backward** button and menu item usually step back one line. **Forward** steps forward one line. **Begin** and **End** move to the start and end of the execution log respectively. **Set beginning** deletes any logged data before the currently shown state. **Play** is equivalent to repeatedly clicking **Forward** until the end of the execution log is reached.

Using the animator scroll bar, you can move to positions in the middle of steps. In this case, **Forward** and **Backward** work like the corresponding animator scroll bar arrows.

In Figure 1, the data view contains (clockwise from top left):

- The container for user-created data (see Subsections 6.3 and 6.2), containing an exception thrown by the user invocation of `SimpleTest.main(String[])`.
- A thread called `PrinterThread` which is defined as a nameless inner class of `SimpleTest`, whose `run()` method has called `SimpleTest.printGibberish(int)`. The numbers after the method names are the invocation id (serial number of the method call) and the line number of the execution position respectively.
- A thread containing the user's invocation of `SimpleTest.main(String[])`. The state of the thread at the time the exception was thrown is shown. The local variables include some strings, a `java.util.HashMap`, and some `java.util.Lists`, which are shown as arrays that have been extracted using the respective interfaces' methods.
- A `SimpleTest` object, containing a field and a description extracted using `toString()`.
- An instance of an unnamed inner class of `SimpleTest`.
- The exception thrown by `SimpleTest.main(String[])`.
- The class `SimpleTest` (which has a single static field).

The data view can be adjusted to suit different use cases. This is discussed further in Section 7.

5.2 The package tree

The package tree shows the packages and classes available to the debuggee VM. It provides access to view settings for classes and packages and allows the user to load classes manually.

The leaf nodes of the tree represent classes. The non-leaf nodes represent packages. By default, all packages (except the root package) are shown minimised. Minimised nodes are shown as triangles, and their subtrees are hidden from view. To minimise or unminimise a package, select **Visualisation/Minimised** from the pop-up menu of the package node.

5.3 Example

An example of the MVT window is shown in Figure 1. The MVT window contains (from top to bottom): menus, animator control buttons and bar, stepping control buttons, the structure panel and the source view with the current line highlighted.

The structure panel contains the data view, the package tree and a user-specified invocation of `setWages` that has not yet been started.

The data view shows a simple database of employees (implemented as an `ArrayList` of `Employees`). The user has invoked a method (`raiseUnionWages`) that raises the wages of every union member in the database by 10% by iterating through the list of employees, calling `raiseWages` on every `Employee` that is a union member, which calls `setWages` to change the wages. A breakpoint has been used to interrupt `setWages` at the point where Peter Jones gets his raise. The execution stack frames corresponding to the methods that are running at this point of execution are shown inside the representation of the thread to which they belong (the innermost call is at the top). Each frame is labelled with the method name, a unique identifier (the number of the call in chronological order) for the call and the current line number. Each frame shown contains some local variables. The data view also shows the `Employee` class, which keeps track of the last (unique, increasing) `id` assigned to an employee.

The package tree shows the classes in the root package and the other packages available to the debuggee VM (the standard Java library and MVT). All packages except the root package are minimised to a single node for clarity.

The invocation structure at the bottom of the structure panel is a temporary structure used to specify the arguments for a method call. When the user chooses a method to execute (from the pop-up menu of an object or class), an invocation structure is shown. When the user has specified the arguments (by dragging values into the invocation structure), he can tell MVT to perform the method call specified by the invocation structure. In this case, the user has already specified the parameter for the `setWages` call, so he can start the method call whenever he likes.

The source code for the example is in the `example` directory in the MVT distribution.

6 Controlling the debuggee

6.1 Loading classes

In order to perform any operations on a class, such as invoking methods defined in it or creating instances of it, the class must be loaded into the debuggee VM. Many standard library classes (such as most of `java.lang`) are automatically loaded on VM startup, and the debuggee VM automatically loads classes when they are required by code that is running on the VM.

If you wish to create instances or invoke methods of classes that have not been loaded automatically by the debuggee VM, you will need to explicitly load them by right-clicking the class in the package tree and choosing **Class/Load Class** from the pop-up menu. This will start a thread in the debuggee VM that loads and prepares the class. As this thread is by default suspended (to allow stepping through any static initialisers that may be executed due to class preparation), you will have to tell the debuggee to continue e.g. by clicking **Continue**. The debugger will stop the debuggee when the class has been loaded and display the class in the data view.

You can create array classes with object or array elements by selecting **Testing/Get array class** from the pop-up menu of the element class. The resulting class is added to the data view. Primitive array classes are available through **Test/Show primitive array class** on the main menu bar.

6.2 Modifying data values

You can set a data value (a primitive value or object reference) to another value by dragging the value or object into the variable you wish to set. If a reference is represented by an arrow, you can drag the end of the arrow to another object of a suitable type to change the reference.

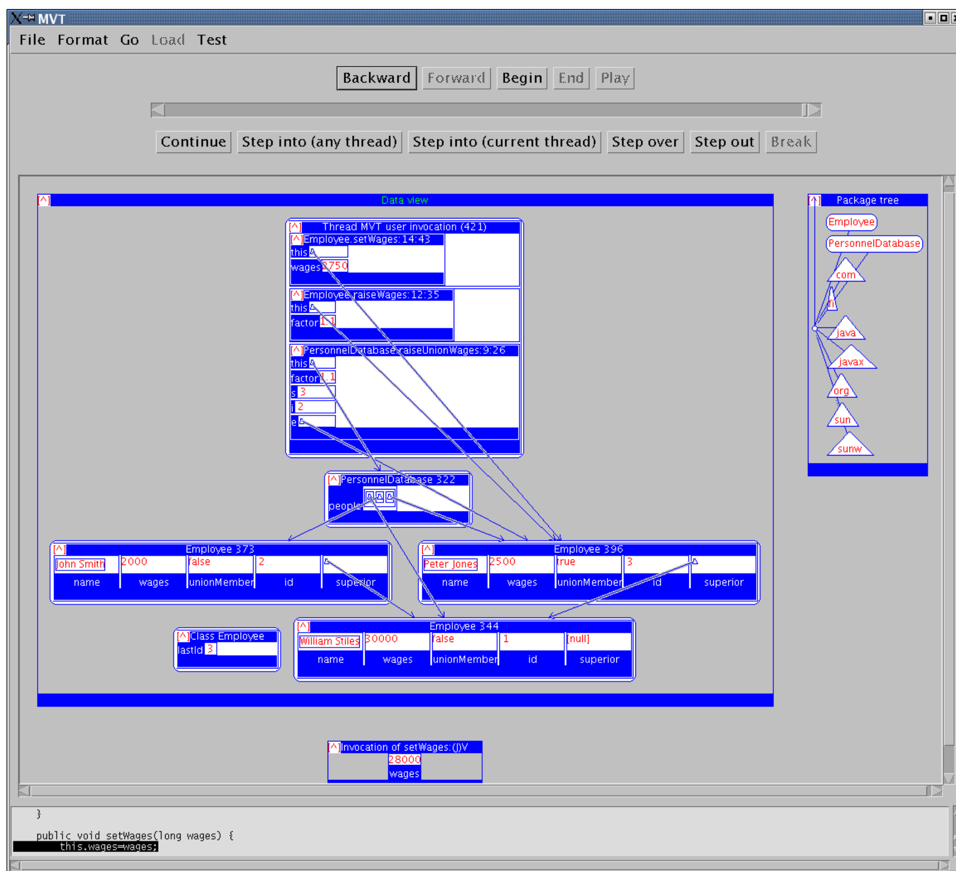

Figure 1: Screenshot of MVT

Code	Type
B	byte
C	char
D	double
F	float
I	int
J	long
L⟨class⟩**;**	Instance of ⟨class⟩
S	short
Z	boolean
[⟨code⟩	array of type with code ⟨code⟩

Table 4: Type codes in signature

To create a new primitive or String value, select the item in the menu **Test/New value** corresponding to the type of the item you wish to create. For boolean values, select the value you want from the submenu. For other values, enter the value you want to create and select **OK**. The new value will be added to the user-created data in the data view.

6.3 Invoking methods

In most cases, you will want to invoke methods and examine their operation and results. In order to execute static methods, right-click the class in the data view and select a method from the **Public methods**, **Protected methods**, **Package methods**, or **Private methods** submenu of the **Testing** submenu of the pop-up menu. This produces a method invocation box (as shown at the bottom of the structure panel in Figure 1). You can specify the parameters for the method invocation by dragging primitive values and objects into the invocation box. The method invocation can then be started by selecting **Execute method** from the invocation's pop-up menu.

Methods are identified by their name and their signature, separated by a colon. The signature consists of the argument types (in parentheses) followed by the return type. Each variable type is encoded as shown in Table 4.

MVT treats constructors as methods that return a new object. The name of the constructor is always <init>.

6.4 Starting, stopping and stepping

MVT has several commands to allow you to control how method execution proceeds. All of them have a command button beneath the animator controls and an item in the **Test** menu with a shortcut key (shown in the menu). They are described in Table 5. There is also a **Break** button that stops the execution of the debuggee.

While the debuggee is running, the MVT user interface (except for the **Break** button) is disabled.

6.5 Setting breakpoints

By selecting **Class/Show source code** from the pop-up menu of a class in the package tree, you can view its source code in a separate window. In this window, you can set and remove breakpoints by clicking on a line in the source code. Lines with breakpoints are highlighted.

Note that source code windows correspond to classes, not to source code files. To set a breakpoint for an inner class, you must open the source code window for the inner class, not the enclosing class, and vice versa.

Command	Effect
Continue	Run the program to the next user method completion, breakpoint or user break command.
Step into (any thread)	Run the program to the next user method completion, breakpoint, user break command or start of execution of a line in an instrumented thread.
Step into (current thread)	Run the program to the next user method completion, breakpoint, user break command or start of execution of a line in the thread whose position is shown in the source view.
Step over	Run the program to the next user method completion, breakpoint, user break command or start of execution of a line in the current method invocation or one enclosing the current invocation in the thread whose position is shown in the source view.
Step out	Run the program to the next user method completion, breakpoint, user break command or start of execution of a line in a method invocation enclosing the current invocation in the thread whose position is shown in the source view.

Table 5: MVT stepping commands

7 Changing view settings

MVT allows the user to select the information to be shown in a variety of ways. The user can choose whether the threads of the program and their stack frames are visible. The user can also show and hide classes and their instances individually, all at once, or all instances of all the classes in a package at once. The user can choose which fields of an object are shown separately for each object or for every instance of a class or interface at once. Referenced objects are always shown if a variable containing a reference is visible.

7.1 Text format

The size and font of the text used in MVT's views can be adjusted using the **Format** menu.

7.2 Global settings

The global view settings can be found in the **Test** menu. You can toggle the visibility of threads and the user data container by selecting **Show threads** and **Show user data** respectively. You can specify the maximum amount of visible objects using the command **Max objects in graph**. The default view properties of each type of data container can be adjusted using the corresponding submenu of the **Test** menu (**Array views**, **Object views**, **Frame views**, **Thread views** or **Class views**). The view settings are shown in Table 6.

7.3 Class settings

Each class has a set of associated properties (shown in Table 6) that describe the desired visualisation. If a view property is not defined for a class, the corresponding property of the most similar (based on defined methods and proximity in the inheritance hierarchy) implemented interface or ancestor class will be used.

To change a view setting of a class, select it from the **Class/Object views** submenu of the pop-up menu of the class in the package tree.

Name	Meaning
FlipX	Should objects be flipped horizontally?
FlipY	Should objects be flipped vertically?
Rotated	Should objects have their X and Y axes swapped?
Nested	Should objects be shown nested inside the referring object, instead of being pointed to by an arrow?
Indexed	Should the names of the fields of this object be shown?
Titled	Should the title of this object be shown?
Rescan	Should extractors be rerun on an object of this class when a method exit on the object is detected?
View field	Comma-separated list of fields to show for objects of this class. Use `!` to view all fields and `,` to view no fields at all. For threads and arrays, index ranges are used instead of field names. A range is either an index or a start index and an end index separated by a colon. If a negative index is specified, the length of the array or amount of stack frames is added to the index (in other words, it counts from the end of the table (`-1`) instead of the start). For example, `1:-2` shows everything except the first and last element.

Table 6: View properties

The **Class** submenu also contains the options **Show all instances** and **Show referenced instances**. These add all instances of a class to the data view and remove all instances except those that are referred to by another object respectively.

7.4 Convertors and extractors

MVT provides data abstraction using *convertors* and *extractors*. Convertors are executed at view generation time and read fields from an object in the data model and generate a data element that is added as a new field to the object. Extractors are executed when a new object is detected and (optionally) when a method call to the object has completed (this is used to detect changes to an object when its implementation is unknown). The values generated by convertors are not stored in the log of data modifications, but the extracted values are. The most commonly used extractor is the *accessor* extractor, which calls an accessor method in the debuggee. The value returned by the accessor (which takes no arguments) is added to the data model as an extra field.

Classes inherit extractors and convertors from their superclasses and implemented interfaces. You can specify the convertors and extractors for a class or interface using the **Edit convertors** and **Edit extractors** menu items in the **Class/Object views** submenu of the pop-up menu of the class in the package tree. This opens an extractor/convertor editing window, that lists the defined extractors or convertors for the class. Clicking **Add** allows you to choose an extractor or convertor to add and the parameters to use with it. The parameters can be strings, `null`, or data objects (selected in the data view using **Test/Select** in their pop-up menu). Clicking **Delete** removes the selected extractor or convertor. Clicking **Close** closes the editing window.

Convertors and extractors are implemented as Java classes in the debugger VM. See the Javadoc documentation for details on implementing convertors and extractors.

7.5 Object-specific settings

The visualisation settings of objects can also be adjusted individually. The **Visualisation** and **Testing** submenus of the pop-up menus of the data containers in the data view contain

the visualisation settings shown in Table 6, except for **Rescan**. Changing these settings using the pop-up menu of an object only affects that object's visualisation.

Classes and threads can be removed from the data view by selecting **Hide** from their pop-up menus. By selecting **Test/Show only when referenced**, an object can be hidden when it is not referred to. If **Test/Show always** is selected, the object is always shown.

8 Exporting and printing views and animations

The views and animations generated by MVT can be exported to several formats. **File/Export/LaTeX** exports the current view in LaTeX 2_ε format. **File/Export/SVG** exports the entire contents of the animator (the whole execution history, unless you have deleted part of it) in SVG format. **File/Export/SVG (compressed)** exports the animator contents in gzip-compressed SVG. Adobe SVG Viewer 3 or later is recommended for the viewing of SVG animations.

In order to print the current view, use **File/Print**. Use **File/Print animation** to print the entire animation as a sequence of images. You can change the page settings for printing using **File/Page setup**.

9 Changes

9.1 Changes in v1.01

- Added startup scripts for Windows.
- Fixed build script problems with example files.
- Fixed problem with JDK install directory name containing spaces.
- Fixed problem with missing source line highlights on Windows.

9.2 Changes in v1.02

- Dragging a string value read by the `StringReader` extractor (as opposed to the `String` object itself) now creates a new `String` with that value instead of failing.
- Disabled mouse events in package tree names to make the menus behave more intuitively. For example, right-clicking on a class name now has the same effect as clicking on the class node border.
- Disabled meaningless drag operations (dragging a class or package, for example).
- Included updated version of Matrix that resolves pop-up menu issues on Java 1.5 on Windows.

10 Known problems and limitations

This chapter describes some known problems and limitations of MVT and how to circumvent them if possible.

10.1 Pop-up menus

Under Linux, the menu items on the top level of the pop-up menu have no effect unless they are left-clicked on Java versions prior to 1.5. This is due to Java bug 4533641. This has been resolved in the XAWT implementation of AWT in Java 1.5.

10.2 Scrolling

Under Linux, the structure panel does not always scroll correctly when dragging or using the mouse wheel. This is due to Java bug 4767797. Use the scroll bars on the structure panel to scroll it instead. Scrolling the structure panel horizontally using its horizontal scroll bar appears to correct this problem temporarily.

10.3 AWT null pointer exceptions

Under Linux Java 1.5 with XAWT, MVT may fail to perform certain operations due to null pointer exceptions in `sun.awt.X11.XMenuPeer.repaintMenuItem`. This appears to be due to Java bug 5106833. Using the Motif implementation of AWT avoids this bug. Popping up the **Test** menu before using any execution control buttons or keyboard shortcuts also appears to prevent the exceptions from occuring.

10.4 Performance issues

Execution of instrumented code in MVT is very slow, due to the large amount of inter-process communication generated by the continuous stream of event notifications required for logging. Extractors make this even worse.

To ensure that MVT runs as quickly as possible, you should not use any unnecessary extractors and only instrument code that you wish to examine. If you are examining a section of a large program, instrument only that section instead of the whole program.

Furthermore, the display of large object graphs is very slow, so you should limit the amount of objects shown at once to something like 500.

Currently, MVT is unsuitable for large test runs due to these performance issues. It may be impossible to solve these problems without extensive redesign of MVT or the creation of an entirely new system.