Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis,
VisSoft 2002, pages 100-109, 2002.
© IEEE Computer Society Press 2002

Reification of Program Points for Visual Execution

Stephan Diehl
Computer Science Department
Saarland University
Saarbrücken, Germany
diehl@acm.org

Andreas Kerren
Computer Science Department
Saarland University
Saarbrücken, Germany
kerren@cs.uni-sb.de

Abstract

Existing reification techniques for Java only allow for inspection and manipulation of Java programs on the class, object and method level, but not at the level of individual program points. In this paper we introduce a reification technique of program points based on source-to-source transformations. Our reification method allows for the association of arbitrary meta-information with program points and to manipulate it during execution. We present examples of the innovative use of such reified program points for visualizing the execution of Java programs.

1 Introduction

Algorithm animation systems are tools to produce and/or interact with dynamic graphical representations of computations. Such animations are meant to support understanding of algorithms and thus are used as teaching aids as well as development tools [12, 21, 7, 5].

Interesting events (IEs) are widely used in algorithm animation systems and usually lead to a system architecture based on the MVC design pattern (model, view, control) which is a combination of the Observer, Composite and Strategy patterns [14]. The program is annotated at important program points with an IE. Whenever execution reaches an IE, information about the current program state is sent to all registered views.

Based on the above mentioned prototypical architecture we developed the GANIMAL framework [9] to ease the creation of interactive algorithm animations. To specify such animations we developed the language GANILA as an extension of Java. It provides a powerful set of features integrating concepts of different classical algorithm animation systems: Interesting events and views (BALSA [4]), step-by-step execution and breakpoints (BALSA-II [2]) and parallel execution (TANGO [20]). In addition it offers new features like alternate interesting events and alternate code blocks, foresighted graphlayout [6], mixing of post mortem and live/online algorithm animation, controlled visualization of loops and recursion, as well as the visualization of invariants for program points and blocks. Controlled visualization allows the execution of certain program points to be visualized, e.g. the last five executions of a loop or every second invocation of a recursive method. Many of these features can be changed at runtime, i.e. the user can select at run time whether two program blocks should be executed in parallel or not, or which of two alternate program blocks should be executed. Most of these features heavily rely on the existence of reified program points. By virtue of reified program points a program can inspect and manipulate its own behavior at the level of program points at runtime.

> *We will use the term* reification *to mean the conversion of an interpreter component into an object which the program can manipulate. One can think of this transformation as converting program (...) into data. We will use the term* reflection *to mean the operation of taking a program-manipulable value and installing it as a component in the interpreter. This is thus a transformation from data to program.*
>
> – D.P. Friedman and M. Wand [13]

The reification technique presented in this paper is based on source-to-source transformations carefully designed to preserve the semantics of the original Java program. Most transformations require previous static analyses of the program including type inference and live variables analysis. The results of these analyses are used to preserve the context (scope of variables and type of expressions) and control-flow when moving code inside a new method. Our compiler called GAJA generates for each program point a method which contains the actual program code for the program point and an object providing meta information about the program point. In this paper we use the term 'settings' to refer to this meta information. Ideally, a reified program

point should be an object containing both settings and program code. This could be achieved by subclassing for each program point an abstract class `ProgramPoint` and overwriting a certain method with the code for this program point. As a consequence the number of generated classes would be proportional to the number of program points. Separating program points and settings allows us to have only a fixed number of classes for settings, while for each class of the source program we generate exactly one target class.

This paper is organized as follows. Section 2 gives examples of settings used in our implementation. Section 3 describes the reification of pure Java programs and Section 4 describes the architecture of our visualization system and the reification of GANILA annotations. In Section 5 some practical results are presented and in Section 6 related work is discussed. Section 7 presents some concluding remarks.

2 Settings

After reification we can associate with every program point arbitrary information and change it at runtime using a GUI (see Figure 1) or whenever the program point is executed. In our current implementation we have the following settings:

- break point: If the current program point is a break point, the execution stops and the user can inspect the program state or resume execution.
- interesting event: The current program point is an interesting event and it can be active or not.
- visits: At each execution of a program point a counter can be initialized or increased to count the number of iterations of a loop or the depth of a recursive call [1]
- invariants: This program point is the start or end of a block for which a certain invariant should be checked, the invariant is added or removed from the invariant view [2].
- parallel execution: This is a virtual program point, its setting determines whether its two blocks are sequentially executed or in parallel.
- alternate: This is a virtual program point, its setting determines which of its two blocks is executed.
- folding: This is a virtual program point consisting of an IE and a block. If this node is active then the IE is sent and none of the IEs contained in the block are used. If it is inactive then the IEs in the block are sent. In either case, the instructions in the block are executed.
- mode: At this program point the animation mode is set to PLAY or RECORD.

Note that we could also store a reference to the abstract syntax tree of the program point in the settings but so far, this has not been not required by our applications.

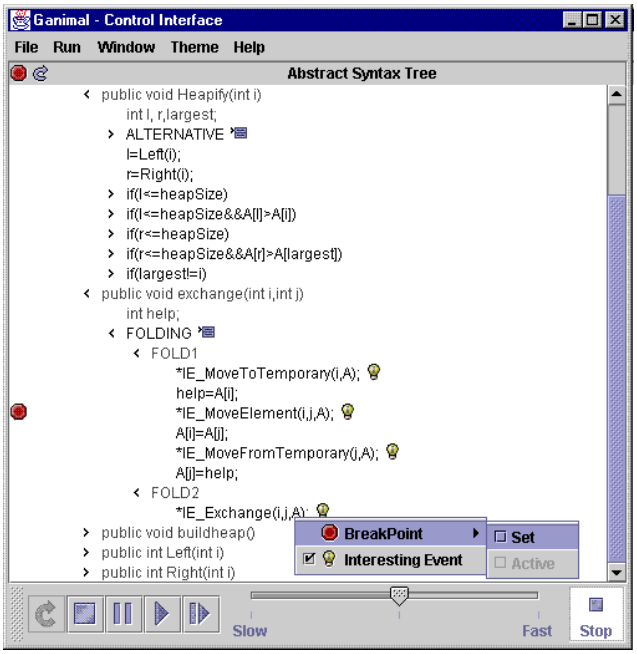

Figure 1. Graphical user interface to change settings at runtime

3 Reification of Java Programs

Reification of program points could be realized in various ways including a meta-interpreter, direct translation into code for the Java Virtual Machine (JVM) or even by extending the JVM.

> *However, it soon became apparent that the JVM and Java were tightly coupled, and any language that compiles into the JVM would lose little in efficiency – and gain much in clarity – by translating into Java as an intermediate stage.*
>
> – M. Odersky and P. Wadler [19]

To illustrate how the GAJA compiler reifies program points we will now look at several example programs. First we discuss the translation of pure Java programs addressing sequences of instructions, method invocations, and control-flow statements. In the next section we look at the translation of the language extensions provided by GANILA. To improve readability, we have removed exception handling code from the generated code shown throughout the paper.

[1] In our current implementation only loop counters are supported.
[2] The invariant view is currently under development.

3.1 Simple Statements

In our first example we define a class `Example1` with a single method `start()`. Both an object variable `x`, that is a non-static variable, and a local variable `y` are accessed in the method body.

```
ganila class Example1 {
  int x;

  public void start(String[] argv) {
    int y = 2;
    x = x + y;     // program point 0
    y = x * y;     // program point 1
  }
}
```

From the code for the class `Example1` we generate a class `Example1Algorithm`. To reify the two program points in this method, we generate a method `dispatch_0()` for the first and `dispatch_1()` for the second program point.

Both variables `x` and `y` must be accessible in the generated method bodies. As the generated methods are defined as object methods in the same class, `x` is still accessible in these methods, whereas the local variable `y` is explicitly passed as an argument to the generated methods. To pass variables from one context to another we provide wrapper classes `Gint`, `Gboolean`, etc. for all primitive types [3]. We use the wrapped values as L-values and unwrapped values as R-values.

By invoking the method `passControlToGUI()` before executing each program point, the GUI gets access to the current state and can intervene. For example it could check wether the current program point is a break point, wait for the user to press a key to proceed execution, or just sleep for a few milliseconds to delay execution. Furthermore it could check whether an invariant still holds and it can highlight the current program point in the code view, a window containing the program code.

```
public class Example1Algorithm
       extends GAlgorithm {
  Gint x = new Gint();

  public void start(String[] argv) {
    Gint y = new Gint(2);
    passControlToGUI(0);
    dispatch_0(y);
    passControlToGUI(1);
    dispatch_1(y);
  }

  public void dispatch_0(Gint a1) {
    x.setValue(x.getValue()+a1.getValue());
  }

  public void dispatch_1(Gint a1) {
    a1.setValue(x.getValue()*a1.getValue());
  }
}
```

[3]The wrapper classes `Integer`, `Boolean`, etc. in `java.lang` are final. Thus we had to define our own wrapper classes to be able to add visualization methods and additional information.

Using the above transformation scheme code for the common dispatch behavior, here calls to `passControlToGUI()` , is repeatedly generated. If we want to change this later on, we would have to change the compiler. To improve maintainability of the compiler we introduce another level of indirection as can be seen in the generated code below. The generated class `Example1Algorithm` inherits the predefined, generic method `dispatch()` from the class `GAlgorithm` which in turn invokes `passControlToGUI()`. To execute a program point, a call to the general method `dispatch()` is generated in place of the original program point.

```
public class Example1Algorithm
       extends GAlgorithm {
  ...
  public void start(String[] argv) {
    Gint y = new Gint(2);
    dispatch(0,new Object[]{y});
    dispatch(1,new Object[]{y});
  }
  ...
}
```

The call `dispatch(`pp`,new Object[]{`$arg_1, \ldots, arg_n$`})` invokes the method `dispatch_`$pp(arg_1, \ldots, arg_n)$ via Java Reflection. If we want to change the dispatch behavior later we no longer have to change the compiler, but we can just change the definition in the class `GAlgorithm`.

```
public class GAlgorithm {
  ...
  public synchronized ControlFlow
         dispatch(int pp, Object[] args) {
    passControlToGUI(pp);
    Class c = getClass(), params[];
    if (args == null) params = new Class[0]
      else params = new Class[args.length];
    params = new Class[];
    for (int i=0;i<params.length;i++) {
      params[i] = args[i].getClass();
    }
    Method m = c.getMethod("dispatch_"+pp,
        params);
```

```
    Object o = m.invoke(this,args);
    if (o instanceof ControlFlow)
      return (ControlFlow) o;
    return new ControlFlow();
  }
}
```

3.2 Control Flow Statements

In the above example we had a simple control flow, just a sequence of two assignments. In the following example we look at method calls with return values and a conditional statement. Other control flow instructions like loops and the `switch` statement are translated in a similar way.

```
ganila class Example2 {
  int x;

  public void start(String[] argv) {
    int y = 2;
    x = max(x,y);      // program point 0
  }

  public int max(int a, int b) {
    if(a > b)          // program point 1
      return a;        // program point 2
    else
      return b;        // program point 3
  }
}
```

Once again we generate methods for each of the four program points. We introduce `ControlFlow` objects to handle `break`, `continue` and `return` statements. Using our reification technique nested blocks are translated into nested calls of `dispatch()`-methods. The execution of one of the statements listed above will terminate the execution of the current block and continue in one of the enclosing blocks. Hence we will refer to this block as the target block. In the translated program termination of a block is implemented by returning from a `dispatch()`-method with a `ControlFlow` object as its return value. All intermediate `dispatch()`-methods just pass this value to their caller until it reaches the translation of the target block. In our example the target block of the `return` statement is the body of the method `max()`. If the `ControlFlow` object represents a return statement, the return is now executed using the value contained in `ControlFlow` object as the return value.

```
public class Example2Algorithm
       extends GAlgorithm {
  Gint x = new Gint();

  public void start(String[] argv) {
    Gint y = new Gint(new Gint(2));
    dispatch(0,new Object[]{y});
  }

  public Gint max(Gint a, Gint b) {
    ControlFlow cfl =
        dispatch(1,new Object[]{a,b});
    if (cfl.isReturn) return (Gint)cfl.value;
  }

  public void dispatch_0(Gint a1) {
    x.setValue(max(x,a1).getValue());
  }

  public ControlFlow dispatch_1(Gint a1,
        Gint a2) {
    if (a1.getValue() > a2.getValue()) {
      ControlFlow cfl = dispatch(2,
          new Object[]{a1});
      return cfl;
    } else {
      ControlFlow cfl = dispatch(3,
          new Object[]{a2});
      return cfl;
    }
    return new ControlFlow();
  }

  public ControlFlow dispatch_2(Gint a1) {
    return new ControlFlow("return",
        new Gint(a1.getValue()));
  }

  public ControlFlow dispatch_3(Gint a1) {
    return new ControlFlow("return",
        new Gint(a1.getValue()));
  }
}
```

It is noteworthy that the `throw` statement does not have to be wrapped in a `ControlFlow` object, because `throw` automatically terminates all intermediate blocks or method invocations until it reaches the matching `catch`-clause.

4 Reification of GANILA Programs

The language GANILA extends Java with annotations to control the visual execution of Java programs. The GAJA compiler generates code which in combination with the runtime system produces the interactive animations. In the generated code all program points of the annotated classes are reified as discussed above.

4.1 Interesting Events

In GANILA we write interesting events as method calls with the prefix `*IE_`. They transfer local information in their arguments to the different views.

```
public boolean isPrime(int n) {
  *IE_Prime(n);
  // rest of method body
  ...
}
```

At runtime instead of the above method call, an event object of class `GEvent` is created. It contains the following information: the current program point, the current animation mode `isRecord()`, the activation state of the event in the GUI `isVisibleEvent(pp)`, the environment for the meta-variables in the controlled visualization of loops and recursion `getLoReDepth()`, the name of the event, and the arguments of the original method call.

```
public Gboolean isPrime(Gint n) {
  dispatch(4, new Object[]{n});
  // translated code for rest of method body
  ...
}

public void dispatch_4(Gint a1) {
  GEvent e;
  e = new GEvent(4,
                 isRecord(),
                 isVisibleEvent(pp),
                 getLoReDepth(),
                 "Prime",new Object[]{a1});
  control.broadcast(e);
}
```

Such a `GEvent` is send to the control object. The control object forwards it to all registered views. Each view can have its own settings and decide whether it will invoke its event handler for this interesting event. So far our events work very much like those in other multi-view event-based algorithm animation systems like ZEUS [3], except for all the additional information stored in the events and the fact that our program points are reified.

As a result our system provides features not present in any of the algorithm animation systems we are aware of [12, 21, 7, 5]. The use and implementation of these features are discussed in the remainder of this section.

4.2 Animation Modes

The GANILA code below shows how to annotate the algorithm to enable post mortem visualization. In particular the recording mechanism allows for the mixing of post mortem and live/online algorithm animation.

```
*RECORD;
  // Code containing IEs to be recorded
*REPLAY;
```

By default algorithms are executed in PLAY mode. In this mode all interesting events are immediately executed. The instruction `*RECORD` selects the RECORD mode. In this mode all interesting events are not executed, but stored by the control object in their dynamic order. The instruction `*REPLAY` first executes all recorded events. Then it switches into PLAY mode.

Many naive post-mortem visualization systems work like this. They just replay recorded events. Although they actually know the whole story before they even draw the first line, they do not exploit this fact to improve the visual output. As a simple example consider a visualization that produces just textual output line by line in a window. If we use online visualization, the window must have a scrolling function. If we use post-mortem visualization looking at the whole text before the first line is printed, the window or font size can be adapted to fit all text onto the screen.

As a consequence recording and replaying events needs some extra effort to enable views to produce better output when using post-mortem visualizations. In our system for each interesting event `*IE_`*eventname* a view has to implement the following methods:

- `IE_`*eventname*`_Play_Visible()` produces visual output and internal state changes.
- `IE_`*eventname*`_Play_Invisible()` produces no visual output, but internal state changes. This method is invoked for deactivated interesting events, such that subsequent active events are executed in a consistent internal state.
- `IE_`*eventname*`_Record_Visible()` produces no visual output, no internal state changes, except for computing additional information used for post mortem visualization. When the event is replayed later on the method `IE_`*eventname*`_Play_Visible()` is invoked. It can exploit this additional information.
- `IE_`*eventname*`_Record_Invisible()` produces no visual output, no internal state changes, except for computing additional information used for post mortem visualization. When the event is replayed later on the method `IE_`*eventname*`_Play_Invisible()` is invoked. It can exploit this additional information.

To relieve the programmer from implementing all these methods, the GAJA compiler produces adapter classes [14] for the views. If the programmer only wants to use online visualization and naive post mortem visualization, he has to write code for the method `IE_`*eventname*`_Play_Visible()`.

To support non-naive post mortem visualization the remaining three methods must be programmed.

We refer the interested reader to our work on Foresighted Graphlayout [6] for more complex examples of non-naive post-mortem visualization using a generic algorithm to draw sequences of evolving graphs while preserving the mental map [18].

4.3 Controlled Visualization of Loops

Next we explain how counting visits of program points and switching between different animation modes can be used to visualize certain iterations of a loop.

Often interesting events are placed within loops or recursive method invocations. If the iteration or recursion is part of a larger algorithm, it can be annoying that all iterations or invocations are visualized. For the user it could be very boring to watch 100 iterations where it could be sufficient for understanding the algorithm to see just the last three iterations. To enable such a selective visualization, GANILA allows to annotate Java's loop statements (`do`, `while`, `for`) with visualization conditions. These are written within brackets following the loop condition:

```
ganila class Example3 {
  public void start(String[] argv) {
    int s = 0;
    for(int j=0;j<100;j++)    // program point 0
        [($n-$i<5)&&($i%2==1)] {
      if(isPrime(j)) {        // program point 1
        s = s + j;
        *IE_Add(s,j);
      }
    }
  }

  public boolean isPrime(int n) {
    *IE_Prime(n);
    ...
  }
}
```

Here the variable `$i` denotes the number of the current iteration and the variable `$n` the maximal number of iterations of the respective loop. Note, that both values can only be computed at run time. In the example those of the last 5 iterations are visualized which are odd numbered, i.e. the event `IE_Prime(95)`, `IE_Prime(97)`, `IE_Add(963,97)` and `IE_Prime(99)`.

The class `Example3` is translated as before, but we also generate special settings for the program point 0, i.e. the one of the loop and extend the general `dispatch()` method. It records all events until the last iteration is reached. Then the value of `$n` is known and it can resend the relevant events. The visualization condition is translated into a test method `checkVisits()` which is part of the settings.

```
st[0] =
  new Settings(0, new Object[]{
    new Visits_Setting(true,
        "($n-$i<5)&&($i%2==1)") {
      public boolean checkVisits(int $i, int $n) {
        return ($n-$i<5)&&($i%2==1);
      }
    }
  });
```

We insert three new method calls into the general `dispatch()` method of Section 3:

```
  public synchronized ControlFlow
      dispatch(int pp, Object[] args) {
    passControlToGUI(pp);
    loopIncrement(pp); // 1st new method call
    ... // compute c and params[] as before
    loopBegin(pp);     // 2nd new method call
    Method m = c.getMethod("dispatch_"+pp,params);
    Object o = m.invoke(this,args);
    loopEnd(pp);       // 3rd new method call
    ... // return as before
}
```

These functions create, delete and increment instances of counters for a given program point. Before we explain these functions in more detail let's look at a more complex example first:

```
void foo() {
  for( ... ) [$n-$i<5]        //program point 0
    { while( ... ) [$n-$i<3]  //program point 1
        { x = x + 5;          //program point 2
          foo();              //program point 3
        }
    }
}
```

Now consider the situation after `foo()` has been recursively called twice and the current program point is 2. At this moment there are 4 live instances of `$i` and `$n`. Thus in our implementation we allocate these variables on a stack, the environment for meta-variables. To compute the value of `$n` we need to record all events, get the value of `$n` after the last iteration and then replay all events. For this, the methods `loopIncrement()`, etc. manipulate the environment as follows:

- `loopIncrement(`pp`)`: Loop counters are incremented at the first program point in a loop. More precisely, if the program point $pp - 1$ is a loop and has a loop condition and the environment is not empty, this method increases the topmost instance of `$i` on the stack.

- `loopBegin(`pp`)`: Unless the program point is a loop with a loop condition, the method does nothing. Otherwise, a new entry for the loop counters is created on the stack. The entry is used to store the values of pp, `$i` and `$n`. If the stack has been empty before, then no other loop condition has been currently active and this method changes the animation mode, such that this and all nested loops are executed in RECORD mode.

- `loopEnd(`pp`)`: Unless the program point is a loop with a loop condition, the method does nothing. Otherwise, it pops the topmost entry off the stack. As described in Section 4.1 every interesting event contains a copy of the current environment for meta-variables where all instances of `$i` are copied by value, while those of `$n` are copied by reference. Thus this method can now set the value of `$n` to the value of `$i`, i.e. its value at the last iteration, and by virtue of the references it changes in all copies of `$n`. If the stack becomes empty, this method lets the control object replay all recorded events. Then the animation mode is set to PLAY.

As a result, when a recorded event is replayed by the control, it contains a copy of the environment with the right values for `$i` and `$n` and can check all loop conditions now. More precisely, only if for each entry (pp, i, n) in this environment the test `st[`pp`].checkVisits(`i,n`)` is true, the event is visibly executed by calling `IE_`*eventname*`_Play_Visible()`, otherwise `IE_`*eventname*`_Play_Invisible()` is invoked.

Controlled visualization of recursion can be implemented in a similar way, here `$n` is the maximal depth of the recursion, i.e. the maximal path length in the dynamic call tree.

4.4 Parallel Blocks

As a final example we look a the parallel operator `*||`. It allows for the execution of two blocks in parallel which is very useful to perform two animations concurrently.

```
ganila class Example4 {
  int x = 1;

  public void start(String[] argv) {
    int y = 2, tmp1, tmp2;
    tmp1 = x;           // program point 0
    tmp2 = y;           // program point 1
      *{ *IE_MoveTo("x","y");
         x = tmp2; *}   // program point 3
    *||                 // program point 2
      *{ *IE_MoveTo("y","x");
         y = tmp1; *}   // program point 4
  }
}
```

In the above program first the two assignments to `tmp1` and `tmp2` are executed sequentially, then the two assignments to `x` and `y` as well as the respective events are executed in parallel. As a result the corresponding animations run in parallel. Note, that if we would use a single auxiliary variable, data dependencies make parallel execution impossible. In other words, the algorithm had to be slightly changed to enable the parallel animations.

To translate this to Java we consider the parallel operator as a program point on its own and translate it into a call to the method `executeParallel()` inherited from the class `GAlgorithm`. In the example the program point of the parallel operator has number 2. For its two blocks we generate the methods `execute_2_1()` and `execute_2_2()`. For simplicity we have removed the translation of the interesting events in the code below.

```
public class Example3Algorithm
       extends GAlgorithm {
  Gint x = new Gint(1);

  public void start(String[] argv) {
    Gint y = new Gint(2),
         tmp1 = new Gint(), tmp2 = new Gint();
    dispatch(0,new Object[]{tmp1});
    dispatch(1,new Object[]{tmp2,y});
    dispatch(2,new Object[]{tmp1,tmp2,y});
  }

  public void dispatch_0(Gint a1)
   { a1.setValue(x.getValue()); }
  public void dispatch_1(Gint a1, Gint a2)
   { a1.setValue(a2.getValue()); }
  public ControlFlow
   dispatch_2(Gint a1, Gint a2, Gint a3) {
    return executeParallel(2,
        new Object[]{a1,a2,a3});
  }
  public void dispatch_3(Gint a1)
   { x.setValue(a1.getValue()); }
  public void dispatch_4(Gint a1, Gint a2)
   { a2.setValue(a1.getValue()); }
  public ControlFlow
   execute_2_1(Gint a1, Gint a2, Gint a3) {
    dispatch(3,new Object[]{a2});
    return new ControlFlow();
  }
  public ControlFlow
   execute_2_2(Gint a1,Gint a2, Gint a3) {
    dispatch(4,new Object[]{a1,a3});
    return new ControlFlow();
  }
}
```

Based on the settings for the program point of the parallel operator `executeParallel()` either creates two threads to invoke the two generated methods in parallel or it invokes them in sequence. In the case of sequential execution the control flow object returned by the code for the first block is first checked, before the code for the second block is executed.

```
public class GAlgorithm {
  ...
  public synchronized ControlFlow executeParallel(
      int pp, Object[] args) {
    currentPP = pp;
    if(st[pp].isParallelActive()) {
      // execute both blocks in parallel
      Thread t1 = new Thread(
          new GAlgThread(this,pp,1,args));
      Thread t2 = new Thread(
          new GAlgThread(this,pp,2,args));
      t1.start(); t2.start();
      t1.join(); t2.join(); }
    } else {
      // execute both blocks in sequence
      Class c = getClass(), params[];
      if (args == null) params=new Class[0];
        else params = new Class[args.length];
      for(int i=0;i<parameters.length;i++) {
        parameters[i] = args[i].getClass();
      }
      Method m =
          c.getMethod("execute_"+pp+"_1",params);
      Object o1 = m.invoke(this,args);
      if (o1 instanceof ControlFlow
          && !((ControlFlow) o1).isEmpty()) {
        return (ControlFlow) o1;
      }
      m = c.getMethod("execute_"+pp+"_2",params);
      Object o2 = m.invoke(this,args);
      if (o2 instanceof ControlFlow)
        return (ControlFlow)o2;
    }
    return new ControlFlow();
  }
}
```

In a similar way alternate blocks and folding of interesting events (see Section 2) are translated into calls to generic `executeAlternative()` respectively `executeFold()` methods.

5 Practical Results

The GANIMAL system comes with a set of predefined views including a GraphView, CodeView, HTMLView for documentation and a SoundView for acoustic feedback. Customized views can be implemented by subclassing. In Figure 1 the abstract syntax of a Heapsort program is shown in the GUI. The user can set break points, select alternate events or alternate code blocks, activate or deactivate interesting events and select parallel or sequential execution of certain blocks. Furthermore the user can control the animation using a VCR like control to start, pause or step through the animation. Figure 2 shows a snapshot of the visual execution of this Heapsort program.

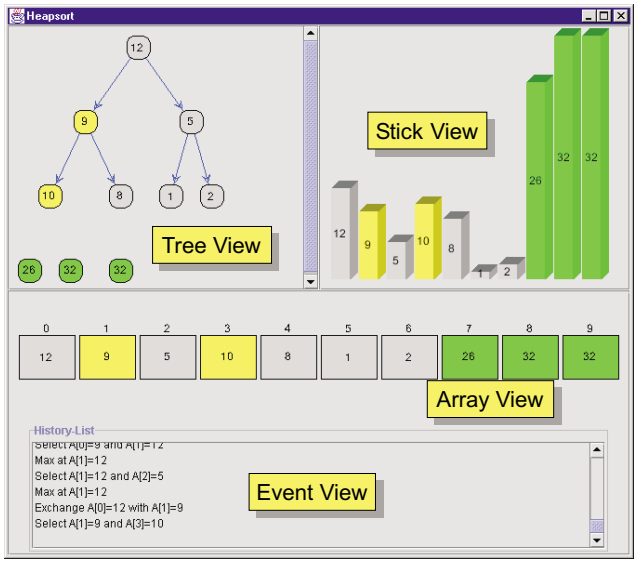

Figure 2. Algorithm animation of Heapsort

We used non-naive post mortem visualization for the interactive visualization of the generation and computation of finite state automata which is an example of the second-order generative approach to explorative learning in computer science [8]. These visualizations have been integrated into an electronic textbook available online [1]. Figure 3 shows the difference between online and non-naive post mortem visualization of the generation of a non-deterministic automaton from a regular expression $(a|b)^*$. In the first row the gradually refined transition diagrams are drawn independently, while in the second row each diagram is drawn using a global layout for the whole sequence. A study with more than 100 students revealed that the electronic text book with its interactive animations was as effective as a classical lecture. Many students reported that the interaction with the system, i.e. entering regular expressions and watching the generation process was highly motivating and helped to dig through the theoretical text [10].

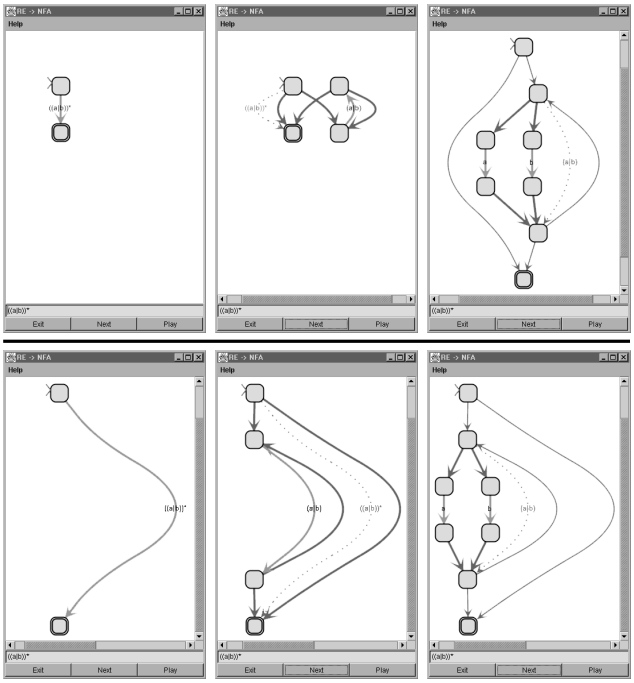

Figure 3. Online and Post Mortem Visualization

6 Related Work

The Java Reflection API provides mechanisms to load, inspect and instantiate classes and to access their methods and variables, even if the names and signatures of these classes, methods and variables have not been known at compile-time. Java Reflection does not provide any mechanisms to access program points at run-time.

The Java Platform Debugger Architecture JDPA [17] consists of an API, wire-protocol and a native interface which Java Virtual Machine implementations must provide for debugging. Using JPDA a debugger gets information like the current stack frame, byte code of methods, sets breakpoints at the level of byte-code instructions and registers for certain events. The JPDA is currently not widely supported by JVMs, not to mention JITs and other native code compilers.

Douence and Südholt [11] propose a generic reification technique based on program transformations. Their method requires the availability of the source code of an interpreter for the language and allows to selectively reify different parts of the interpreter, presumably also those parts interpreting individual program points.

Lawall and Muller [16] augment programs with calls to a generic function `checkpoint()` which stores the current state of certain objects for later recovering from errors or post mortem inspection. To improve performance they use automatic program specialization to generate specialized checkpointing routines which do not traverse and store data, for which static analyses have determined that they have not been modified since the last check point.

7 Concluding Remarks

Our reification technique allows to access individual program points at runtime. Arbitrary meta-information can be associated with these program points and manipulated during execution. The execution behavior can be extended or modified by changing the generic `dispatch()` method. We discussed several extensions which we used for visual execution of Java programs, most notably the mixing of online and post-mortem visualization and the controlled visualization of loops and recursion.

The reification method presented in Section 3 could also be used for different purposes. For example, we could introduce another level of indirection and have a table `PP` that maps numbers to program points. Then the call `dispatch(`pp`, ...)` would invoke the method `dispatch_`$pp'(arg_1, \ldots, arg_n)$ where $pp' =$ `PP[`pp`]`. By changing entries in this table, e.g. using a GUI, the user could build a program from fragments at runtime. This could be useful in a constructivist approach to teaching algorithms.

Besides the above application of reified program points for visual execution of Java programs, our reification technique could also be interesting for software development. As reflection is a key mechanism for late-composition and thus component software [22], the question of what role reified program points could play in this context is left as future research.

A prototypical implementation of the compiler, as well as interactive animations including Heapsort and the generation and computation of finite automata are available. More information about the GANIMAL project, as well as more examples can be found online [15].

8 Acknowledgements

We would like to thank Carsten Görg. This research has been partially supported by the German Research Council (DFG) under grant WI 576/8-1 and WI 576/8-3.

References

[1] Beatrix Braune, Stephan Diehl, Andreas Kerren, Torsten Weller, and Reinhard Wilhelm. Generating Finite Automata – An Interactive Online Textbook. `http://www.cs.uni-sb.de/GANIMAL/GANIFA`.

[2] Marc Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.

[3] Marc Brown. Zeus: A System for Algorithm Animation and Multiview Editing. In *IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[4] Marc Brown and Robert Sedgewick. A system for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984.

[5] Stephan Diehl, editor. *Software Visualization*, Springer State-of-the-Art Survey LNCS 2269. Springer Verlag, 2002.

[6] Stephan Diehl, Carsten Görg, and Andreas Kerren. Preserving the Mental Map using Foresighted Layout. In *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization VisSym'01*, 2001.

[7] Stephan Diehl and Andreas Kerren, editors. *Proceedings of the GI-Workshop "Software Visualization" SV2000*, Technical Report A/01/2000, FR 6.2 Informatik, University of Saarland, May 2000. `http://www.cs.uni-sb.de/tr/FB14`.

[8] Stephan Diehl and Andreas Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001*. ACM, 2001.

[9] Stephan Diehl, Andreas Kerren, and Carsten Görg. Visualizing Algorithm Live and Post Mortem. In *Software Visualization*, Springer State-of-the-Art Survey LNCS 2269. Springer Verlag, 2002.

[10] Stephan Diehl, Andreas Kerren, and Julia Kneer. Evaluation of the Educational Software GANIFA (in German). `http://www.cs.uni-sb.de/GANIMAL/GANIFA/evaluation.pdf`.

[11] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1):7–34, 2001.

[12] Peter Eades and Kang Zhang, editors. *Software Visualization*. World Scientific Pub., Singapore, 1996.

[13] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, Austin, Texas*, pages 348–355. ACM, 1984.

[14] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[15] Ganimal. Project Homepage. http://www.cs.uni-sb.de/GANIMAL, 2000.

[16] Julia Lawall and Gilles Muller. Efficient Incremental Checkpointing of Java Programs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*. IEEE Computer Society, 2000.

[17] Sun Microsystems. *JavaTM Platform Debugger Architecture*, 2001. http://java.sun.com/products/jpda.

[18] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[19] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[20] John Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9), 1990.

[21] John .T Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization*. MIT Press, 1998.

[22] Clemens Szyperski. *Component Software*. Addison-Wesley, 1998.