Animating Algorithms Live and Post Mortem

Stephan Diehl, Carsten Görg, and Andreas Kerren

University of Saarland,
FR 6.2 Informatik,
PO Box 15 11 50,
D-66041 Saarbrücken, Germany
{diehl,goerg,kerren}@cs.uni-sb.de

Abstract.

We first give an overview of the features of the GANIMAL Framework introducing several new concepts not present in any previous algorithm animation system. Then we focus on its mechanisms for mixing live and post mortem visualization which are in particular very useful for algorithms which restructure graphs.

1 Introduction

In recent years we have developed several educational software systems for topics in compiler design and theoretical computer science [1,13]. These systems have in common that they teach computational models by animating computations of instances of these models with example inputs.

In the project GANIMAL we develop generators, which produce interactive visualizations and animations of different compiler phases. The generators form the basis of new kinds of exercises as part of educational software [9,8]. The learner can focus on certain aspects in the generated, interactive animation and see what effects small modifications in the specification have. With the help of such observations he formulates hypotheses and checks these empirically. The learning software does not act as an anonymous, all-knowing authority which shows his errors. Instead, our approach offers a way for explorative, self-controlled learning. Such a visual experimental approach is not meant to replace, but to enhance classical teaching of theoretical contents.

To ease the creation of interactive animations we developed the GANIMAL Framework. The GANIMAL Framework and in particular the language GANILA provide a powerful set of features. It integrates concepts of different classical systems: Interesting events and views (BALSA [3]), step-by-step execution and breakpoints (BALSA-II [2]), and parallel execution (TANGO [15]). In addition it offers new features like alternative interesting events and alternative code blocks, visualization of invariants for program points and blocks, foresighted graphlayout, and mixing of post mortem and live/online algorithm animation which is a prerequisite for visualization control of loops and recursion, i.e. the

S. Diehl (Ed.): Software Visualization, LNCS 2269, pp. 46–57, 2002.

ability to visualize only the execution of certain program points, e.g. the last five executions of a loop or every second invocation of a recursive method.

This paper is organized as follows. Section 2 introduces the GANIMAL framework, i.e. the software architecture and the basic workflow for creating animations. Section 3 describes the annotations provided by the GANILA language for live algorithm animation and Section 4 describes those for inserting post mortem visualizations into live animations and based on this the visualization control for loops and recursion. As an example we compare in Section 5 live and mixed mode animations of an algorithm which computes least upper bounds. Section 6 concludes.

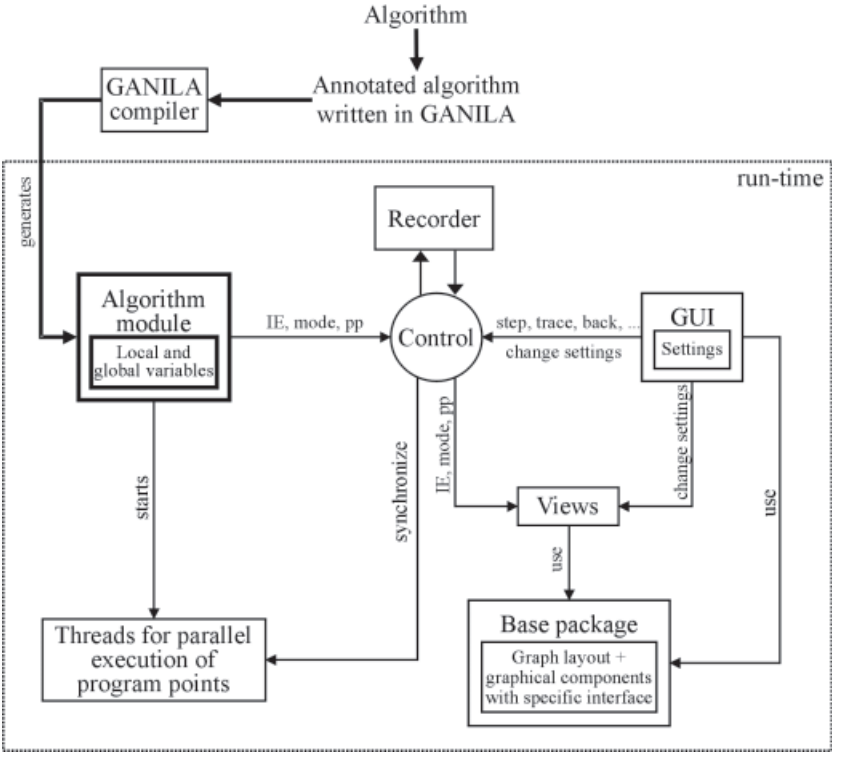


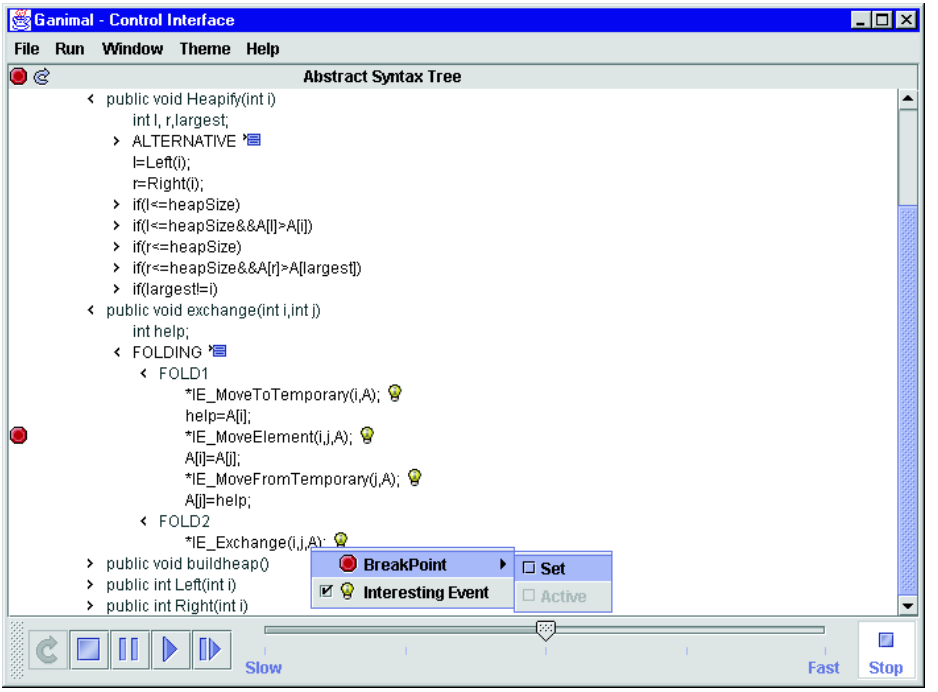
Fig. 1. The GANIMAL Framework

2 The GANIMAL Framework

Based on GANIMAM [10] and the experiences found in related work [6,16] we designed the GANIMAL Framework, see Figure 1. It consists of the GANILA Compiler and a runtime system. The compiler generates code which in combination with the runtime system produces the interactive animations. More

precisely, given a specification written in the language GANILA, the generator produces an algorithm module and the initial settings, i.e. meta-information associated with each program point of the algorithm. During the execution of the algorithm this module sends interesting events (`IE`) containing the current program point `pp` and the current animation mode (RECORD,PLAY) to a control object. The control object checks the settings for this program point. If the interesting event has not been deactivated at this program point, it is send to all views. Each view can have its own settings and decide whether it will invoke its event handler for this interesting event. Based on the animation mode the event handling routines produce graphical output or simply change some internal state and defer the graphical output until the mode is set to PLAY. At runtime the graphical user interface (Figure 2) can be used to change the settings of each program point.

All views should use the base package which consists of a set of Java classes providing primitive methods for communication, graphical output, and animation. The use of the base package fosters a consistent look-and-feel of different views.



At run-time the user can set break points, select alternative events or alternative code blocks, activate or deactivate interesting events, and select parallel or sequential execution of certain blocks. Furthermore he can control the animation using a VCR like control to start, pause, or step through the animation.

Fig. 2. Graphical User Interface

3 Annotations for Live Animation

The language GANILA extends Java by interesting events, parallel execution of program points, recording and replaying mechanisms (e.g. by foresighted graph layout), break- and backtrack points, and declarations to import views. The compiler called GAJA translates GANILA into Java. For every annotated program point its annotation can be activated and deactivated at run time using a graphical user interface. The resulting settings can be defined for the whole animation, as well as individually for each view.

3.1 Predefined Views

Our system provides a set of predefined views which can be imported into a GANILA program using `view <Name>(Parameter)`.

```
// A view without parameters
view CodeView();
// A view with parameters
view SoundView("http://www.cs.uni-sb.de/sounds/");
// An algorithm-specific view
view HeapsortView();
```

The developer of the animation can simply use these views or extend their functionality using inheritance. The methods of the views are event handling routines. In the example above `HeapsortView` is such a newly created view.

HTMLView: In GANILA it is possible to associate program points with web pages. A third party component integrated into the system (IceBrowser-JavaBean) allows to show HTML content in a view. Moreover it is possible to transfer runtime data, which is accessible at the program point, to a CGI-Script on a server. The server can thus produce context-sensitive HTML pages. In "Literate Programming" [14] a static document is produced from the documentations at different program points in the source code. In contrast, in GANILA documentation can be shown whenever the program point is reached during execution.

GraphView: The GraphView provides several algorithms to layout a graph, see Section 4.2 for more details. Nodes and edges can be added or removed. Here almost all kinds of Java SWING components can be used as nodes.

CodeView: The CodeView shows a textual representation of the program executed and highlights the current program point.

SoundView (Aura): Analogous to the HTMLView, this view associates program points with sound files, e.g. containing spoken explanations. As part of an interesting event it receives the URL of a sound file. Starting, stopping, and repeating the play-back, as well as its volume can be controlled by interesting events.

3.2 Interesting Events

The following excerpts show the GANILA specification of a simple operation, which is used as an example by various algorithm animation systems: the swapping of the content of two elements of an array, here `a[i]` and `a[j]`.

```
help = a[i]; *IE_MoveToTemporary(i);
a[i] = a[j]; *IE_MoveElement(i,j);
a[j] = help; *IE_MoveFromTemporary(j);
```

Interesting events have the prefix `*IE_` and transfer local information in their arguments to the different views. Obviously in such a view the value of `a[i]` could be moved to a representation of the auxiliary variable. Then the value of `a[j]` would be moved to `a[i]` and finally the value of the auxiliary variable would be moved to `a[j]`. So far, the GANILA events (GEvents) work very much like those in other multi-view event-based algorithm animation systems like ZEUS [4]. One important difference is that such a GEvent is first send to the control of the framework and is subject to the settings like every program point. As a consequence the user can activate or deactivate the effect of an interesting event at run time using the GUI, see Figure 2. The event handlers of each view must be programmed such that they actually receive a deactivated event and they might even change the internal state of the view to prevent inconsistencies, but it should not produce any visual output. Every view registers with the control object which in turn forwards each event to all registered views. An event handler can even create new views which it can register with the control object. The algorithm object, the control object, and the views are implementing the MVC design pattern (model, view, control) which is a combination of the Observer, Composite, and Strategy patterns [11].

3.3 Alternative Interesting Events and Alternative Blocks

GANILA also supports the grouping of program points by enclosing them in `*{` and `*}` to from a block. The statement `*FOLD *{ <Eventlist> *}` triggers one or more alternative GEvents for a program point or block. The following example also shown in the GUI in Figure 2 illustrates the use of this statement:

```
public void exchange(int i, int j) {
    int help;
     *{  help = A[i]; *IE_MoveToTemporary(i,A);
         A[i] = A[j]; *IE_MoveElement(i,j,A);
         A[j] = help; *IE_MoveFromTemporary(j,A);
      *}
     *FOLD *{ *IE_Exchange(i,j,A); *}
}
```

Using the GUI the user can decide at run time whether the events in the block or the alternative event is triggered. In both cases the program code in

the block is executed. By selecting the alternative event the views could move the two values of the field in parallel to their new positions. Note that in this solution the event handler could use concurrency internally. This is completely different from using the parallel operator as discussed in the next section.

The `*FOLD` construct is meant to support semantical zooming, i.e. in many cases the events in the block, in particular if other methods are invoked, will produce more fine grained animations than the alternative events.

In contrast to `*FOLD` the GANILA construct `*ALT` allows the programmer to provide two different program blocks which should produce the same results. The user can then decide in the GUI which of these program blocks should be actually executed.

```
int min;
*{ min=a[0];
   for(int i=1;i<a.length;i++)
    { *IE_Compare(a,i,min);
      if (a[i]<min) min=a[i];
    }
*}
*ALT
*{ min=a[a.length];
   for(int i=a.length;i>=0;i--)
    { *IE_Compare(a,i,min);
      if (a[i]<min) min=a[i];
    }
*}
```

3.4 Parallel Execution

Using the operator `*||` two program points or blocks can be executed in parallel.

```
    *{ *IE_AssignTemporary(1,i); help1 = a[i]; *}
*|| *{ *IE_AssignTemporary(2,j); help2 = a[j]; *}

    *{ *IE_MoveTemporary(j,1); a[j] = help1; *}
*|| *{ *IE_MoveTemporary(i,2); a[i] = help2; *}
```

In the above program first the two assignments to `help1` and `help2`, as well as the respective events are executed in parallel, then the two assignments to `a[i]` and `a[j]` and 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. The parallel operator automatically creates, starts and synchronizes Java threads for each of the two blocks.

3.5 Test of Invariants

To understand an algorithm it is often necessary to look at properties, which are true for all program states at certain program points. In our framework the developer of an animation can provide a hypothesis and have it checked at certain program points. In the following example the so-called heap property is checked for a part of the heap sort algorithm:

```
*IV(a[i]>=a[2*i+1] && a[i]>=a[2*i+2])
*{
   // part of the heap sort algorithm
*}
```

If the expression is an invariant of a program point, then it should never yield `false` when this program point is executed. If a block is annotated with such an expression, the user will see which program points change the program state such that the invariant is violated, and which program points reestablish the invariant. In addition the user can formulate hypotheses at run time and have them tested by the system. In doing so it is sometimes necessary to invoke complex functions, which have been programmed by the developer of the animation. As it does not make sense to enable the user to invoke every function of the program, the developer can annotate those functions with `interactive` which should be accessible through the GUI at run time.

```
interactive boolean heapProperty(a,i) {
   // checks the heap property
   return a[i]>=a[2*i+1] && a[i]>=a[2*i+2];
}
```

Invariant visualization in GANILA is an example of state mapping [5], i.e. the visualization is not triggered at certain program points through events, but the view has direct access to the program state and automatically adapts its visualization whenever the state changes.

3.6 Break and Backtrack Points

Program points can be marked with `*BREAK` in the GANILA code or through the GUI at run-time as break points. When the execution of the algorithm reaches this program point, the execution is paused and the user can investigate the current state, continue with the animation, or trace it step-by-step.

Backtrack points are marked with `*SAVE`. When the execution of the algorithm reaches such a program point, the current state is copied to the history. Backtrack points are a means to implement reverse execution of the algorithm or repeated execution from a certain point with changed settings. Another way to repeat the execution is to replay all interesting events. This is a more time-consuming, but less memory-consuming alternative provided by the system.

4 Mixing Live and Post Mortem Visualization

In addition to sending events to all registered views the control object can record all events and resend them later. In this case the event handling routines of each view produce no graphical output, but can change some internal state and defer the graphical output until the event is resend. In this section we look at those constructs of GANILA which enable mixing of live and post mortem visualization.

4.1 RECORD/REPLAY

The GANILA code below shows how to annotate the algorithm to enable post mortem visualization. In

```
*RECORD;
  // annotated program code, e.g.
  // for the generation of an NFA from a regular expression
*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.

To enable views to interpret interesting events being fully aware of what events will occur next, the control also forwards events in RECORD mode to all views, but the views are only allowed to modify their internal state, but no graphical output must be produced. This must be deferred until the recorded events are resend.

The `*RECORD/*REPLAY` mechanism allows to mix post mortem and life/online algorithm animation. This is a feature not present in any of the algorithm animation systems we are aware of.

4.2 Foresighted Graphlayout

Often animations of algorithms which change graphs are confusing because they add or remove nodes and edges, and as a consequence the layout of the whole graph is recomputed. In the new layout nodes are drawn at new positions, and a smooth animation called morphing moves nodes from their old to their new positions. Such animations are often nice to look at, but for the user it is not apparent which modifications are due to the animated algorithm and which are due to the drawing algorithm. GANILA supports mechanisms for foresighted

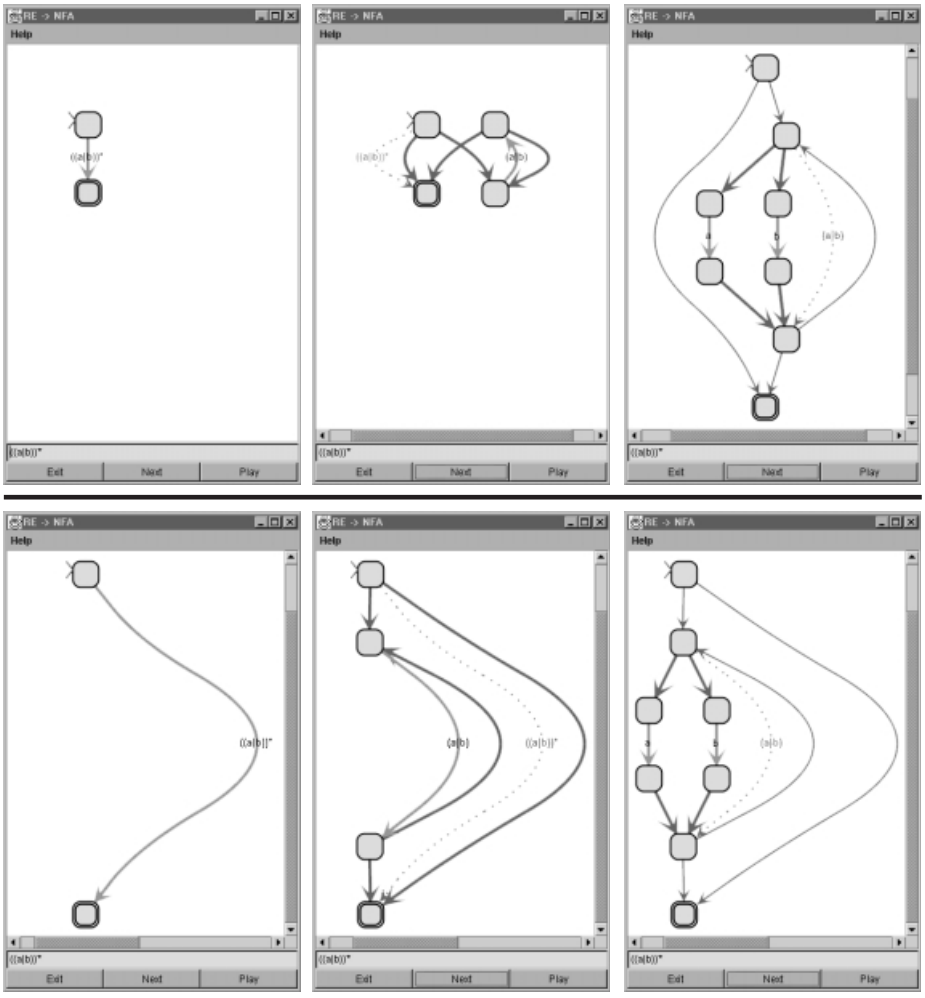


Fig. 3. Ad-Hoc (upper row) and Foresighted Graphlayout (lower row) animating the generation of finite automata

layout, i.e. a graph is drawn exploiting information about subsequent changes of the graph [7].

Figure 3 illustrates how the mechanism can be used to animate the generation of a nondeterministic finite automaton from a regular expression (RE→NFA). The GraphView automatically uses Foresighted Layout when events are recorded and replayed. In the upper row three generation steps are shown using a usual graph drawing algorithm; in the row below Foresighted Layout is used. Without Foresighted Layout it is difficult to see which nodes and edges are added or removed at each step.

4.3 Controlling the Visualization of Loops and Recursion

Often interesting events are placed within loops or recursive method invocations, e.g. when a list is traversed by an iterative sorting algorithm like insertion sort or a recursive sorting algorithm like Quicksort. 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 and it could be sufficient for understanding the algorithm to see just the last three iterations. Our solution to this problem is based on recording all and replaying only certain events at the end of the loop or recursion. 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:

```
for(int j=0;j<100;j++) [$i>=$n-5] { foo(j); }
```

The animation of the execution of the above example program will only visualize the last five invocations of the function `foo()`. 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.

Analogous to the annotation of loops recursive method invocations can be annotated. Here the variable `$i` represents the current depth and the variable `$n` the maximal depth of the recursion.

Animation control for loops and recursion first records all events until the last iteration or recursion is reached. Then it know the value of `$n` and can resend the relevant events.

5 Example: Animating the Computation of Least Upper Bounds

To illustrate the advantages of mixing live and post mortem visualization we look at an algorithm for computing a complete semi-lattice given a set of pairs of integers. A complete semi-lattice contains for each two pairs (a, b) and (a', b') their least upper bound $(\mathsf{max}(a, a'), \mathsf{max}(b, b'))$. An example animation for the set $\{(2, 1), (3, 1), (1, 4)\}$ is shown in Figure 4. After step 10 the user adds interactively the pair $(1, 2)$ to the initial set of pairs. To produce the animation we record all events before the user interaction. Then we replay these using adhoc (upper row) or Foresighted (lower row) Layout. Now the user sees the actual state (step 10) and can change the state before the animation continues. In this example we actually only need the simplest version of foresighted layout. In the adhoc layout at almost every step nodes and edges change their positions; intermediate morphing animations help the user keep track of the mental map. Using Foresighted Layout this is only the case between step 10 and 11, because we cannot foresee the result of the user interaction. At all other steps no position changes of nodes and edges take place. At step 21 only an edge between the pair

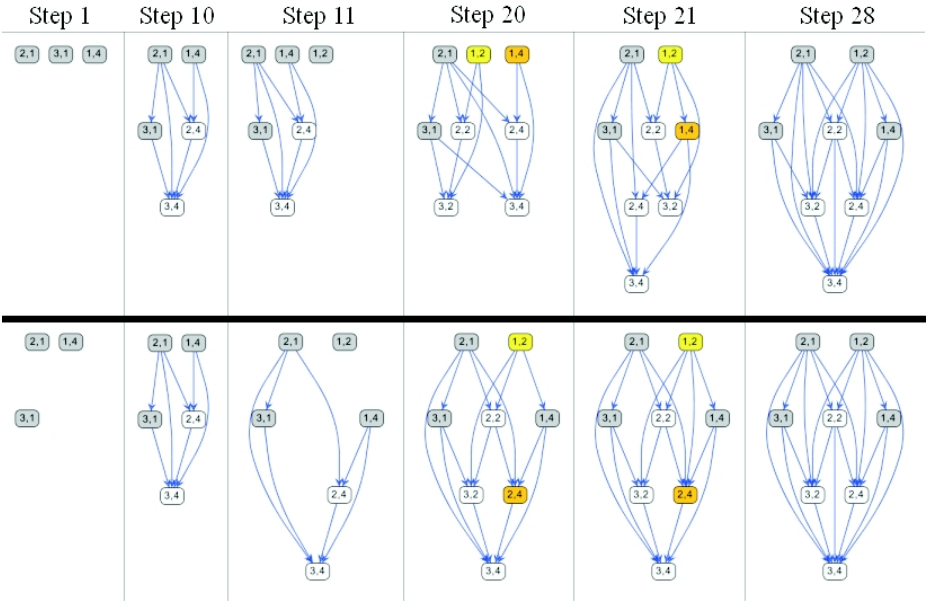


Fig. 4. Adhoc (upper row) and Foresighted Graphlayout (lower row) animating the computation of least upper bounds

$(1, 2)$ and $(2, 4)$ is added. As a consequence adhoc layout changes the position of almost every node, whereas Foresighted Layout just adds this edge.

6 Conclusion

A prototypical implementation of the compiler, as well as interactive animations, which have been produced by the compiler (e.g. heap sort, 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 [12].

Acknowledgement. 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. B. Braune, S. Diehl, A. Kerren, and R. Wilhelm. Animation of the Generation and Computation of Finite Automata for Learning Software. In *Proceedings of Workshop on Implementing Automata*, volume Springer LNCS 2214, Potsdam, 2001.
2. M. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.

3. M. Brown and R. Sedgewick. A system for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, Minneapolis, MN, 1984.
4. M. H. Brown. Zeus: A System for Algorithm Animation and Multiview Editing. In *IEEE Workshop on Visual Languages*, pages 4–9, 1991.
5. Camil Demetrescu, Irene Finocchi, and John Stasko. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001.*
6. Eds.: S. Diehl and A. Kerren. 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`.
7. S. Diehl, C. Görg, and A. Kerren. Preserving the Mental Map using Foresighted Layout. In *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization VisSym'01*, 2001.
8. S. Diehl and A. Kerren. Increasing Explorativity by Generation. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000.* AACE, 2000.
9. S. Diehl and A. Kerren. Levels of Exploration. In *Proceedings of the 32nd Technical Symposium on Computer Science Education, SIGCSE 2001.* ACM, 2001.
10. S. Diehl and T. Kunze. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems*, 16(7), 2000.
11. 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.
12. Ganimal. Project homepage. `http://www.cs.uni-sb.de/GANIMAL`, 2000.
13. A. Kerren. Animation of the Semantical Analysis. In *Proceedings of 8. GI-Fachtagung Informatik und Schule INFOS99 (in German)*, Informatik aktuell. Springer, 1999.
14. D. E. Knuth. *Literate Programming.* Center of the Study of Language and Information - Lecture Notes, No. 27. CSLI Publications, Stanford, California, 1992.
15. J. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9), 1990.
16. J. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In *Proceedings of the 1998 ACM SIGCSE Conference*, San Jose, CA, 1997.