A Rule-based Approach for Animating Java Algorithms

Nada Sharaf, Slim Abdennadher
Computer Science and Engineering Department
The German University in Cairo
Cairo, Egypt
{*nada.hamed, slim.abdennadher*}*@guc.edu.eg*

Thom Frühwirth
Institute of Software Engineering and Compiler Construction
University of Ulm
Ulm, Germany
thom.fruehwirth@uni-ulm.de

***Abstract*—Over the past years, visualization of programs has been widely applied. Algorithm animation was proven to aid in teaching and learning. It provides a convenient medium for beginners to a programming language by giving them the ability to visually discover how their programs are running. It also provides experts of a language with a means to have a visual trace utility. Lately, a new approach for adding visualization features into Constraint Handling Rules (CHR) programs was proposed. The new methodology was a dynamic one able to animate different types of algorithms. The work in this paper aims at introducing a revised extension that is able to embed visualization features into Java programs. With the new extension, Java algorithms could be animated without the need of doing any modifications to the code. In addition, the provided technique is still a general one able to animate different kinds of algorithms.**

***Keywords*-Java Programs, Visual Language, Constraint Handling Rules, Algorithm Animation**

I. Introduction

Algorithm animation has been adopted by programmers and new comers of programming languages. It allows humans to visually perceive how the algorithms work. Animation typically shows the different states of the data structures associated with an algorithm [1].

Animation tools have been proved to increase the effectiveness of learning [2]. The study in [2] was done with "Jeliot", a tool for animating Java programs. The study showed that a notable percentage of students had better results. In [3], the introductory programming course was re-defined using a visual approach. The authors reported an increase in the pass rates from 12% to 23% compared to the previous year. In [4], a meta-study of 24 experimental studies was performed. Through the analysis done, it was found that algorithm visualization was educationally effective.

Over the years different animation systems were proposed. Java, a popular object-oriented programming language, has a lot of associated tools. BlueJ [5] is a system available for Java programmers and students. It presents the structure of the available classes in a Unified Modeling Language (UML) diagram. Users are able to interact with the available classes and objects. With Greenfoot [6], students are able to generate 2D graphical applications. The classes used in every scenario are visualized. The system has two main classes: *World* and *Actor*. Each user scenario should have at least one subclass for each. Actors are the objects available in the *world*. Actors have the necessary behavior to implement the needed scenario. Each actor class is assigned an image thus producing the animations/simulations. Alice [7] enables its users to add 3D animations. It offers users with different 3D models (Java classes). Instances of such classes populate the virtual world of the animation.

The previous systems, however, do not relate the animations to the algorithms being executed. They could be thus helpful to learn about problem solving or Java in general. However, in a programming course, the animations produced should relate to the implemented algorithm to allow students to visually depict the data structures affected by their code. There are some tools that focused on the executed algorithms. Jeliot [8] is a system aiming at offering its users with visualizations of their Java programs. Jeliot is able to provide a visual view of the executed Java programs including all their details such as variables, method calls, etc. Although Jeliot showed the effect of executing the algorithm on the different aspects of the program, it was still not able to visually show users a visual representation of the data structures being affected. VisuAlgo[1], on the other hand, provided examples that offer animations for different kinds of algorithms. The animations visualize the associated data structures and show the changes occurring to them. The problem with VisuAlgo is the fact that the data is static. Users are not able to add their own algorithms to trace or visualize them. The scripting language Animal [9] could be used to generate the needed animations. However, users would need to manually embed visual objects into the code which would result in a non-generic approach. Users could use the associated JavaAPI. However, they have to know many details about the produced visual objects to be able to use them correctly. A similar problem occurs with Balsa [10] and Zeus [11]. Both systems use the notion of interesting events. However, users need to know many details to be able to use them. Java is thus lacking a general purpose animation tool that is able to provide its users with animation features for any type of algorithm. The aim of the work presented in

[1]http://visualgo.net/

this paper is to propose a generic approach for embedding visualization features into Java programs to animate different kinds of algorithms. With this new approach, users do not have to do any changes to the code or know any details about the animation objects.

The paper is organized as follows: Section 2 provides a background over Constraint Handling Rules (CHR) and an overview of annotation rules. Section 3 shows the extension of annotation rules for Java programs. Section 4 shows how the new Java programs work. Section 5 shows in more details how the visual objects are added. Section 6 provides a running example. Finally conclusions and directions of future work are discussed.

II. Background

In Balsa [10] and Zeus [11], the notion of interesting events for animation was used. This notion was extended in [12] to provide a general purpose animation tool for programs implemented through Constraint Handling Rules (CHR) [13]. This section introduces details of CHR programs and how they work [14]. In addition, annotation rules used for animating CHR programs are discussed.

CHR [15] has developed over the years into a general purpose language. A CHR program consists of rules that rewrite constraints in the store until a fixed point is reached where no more rules are applicable. A CHR program could contain user-defined/CHR constraints in addition to built-in constraints (handled by the host language). The constraint store is initialized through the query of the user. A CHR rule has a head, a body and an optional guard. The head should contain CHR constraints only. The guard should only consist of built-in constraints. However, the body could contain CHR and built-in constraints. There are three types of CHR rules: propagation, simplification and, the more general, simpagation rules. A rule is executed if the constraint store contains constraints that match the head of the rule and if the guard is satisfied [16]. A propagation rule has the following form:

$$optional\_rule\_name \; @ \; H_k \; ==> \; Guard \mid Body.$$

A propagation rule adds to the constraint store the constraints in the body of the rule.
A simplification rule has the following form:

$$optional\_rule\_name \; @ \; H_r \; <=> \; Guard \mid Body.$$

A simplification rule removes the matching head constraints from the constraint store and adds the body constraints.
A simpagation rule has two types of head constraints: constraints to be kept and constraints to be removed. It has the following format:

$$optional\_rule\_name \; @ \; H_k \setminus H_r \; <=> \; Guard \mid Body.$$

On executing a simpagation rule, constraints matching H_k are kept in the constraint store and constraints matching H_r are removed. Simpagation rules are considered the most general CHR rule type. A propagation rule is a simpagation rule with an empty H_r. A simplification rule, on the other hand, is a simpagation rule with an empty H_k. The following is an example of the `leq` solver in CHR. `leq(A,B)` means that `A` is less than or equal to `B`.

```
idempotence  @ leq(A,B) \ leq(A,B)
                          <=> true.
reflexivity  @ leq(A,A) <=> true.
antisymmetry @ leq(A,B), leq(B,A)
                          <=> A=B.
transitivity @ leq(A,B), leq(B,C)
                          ==> leq(A,C).
```

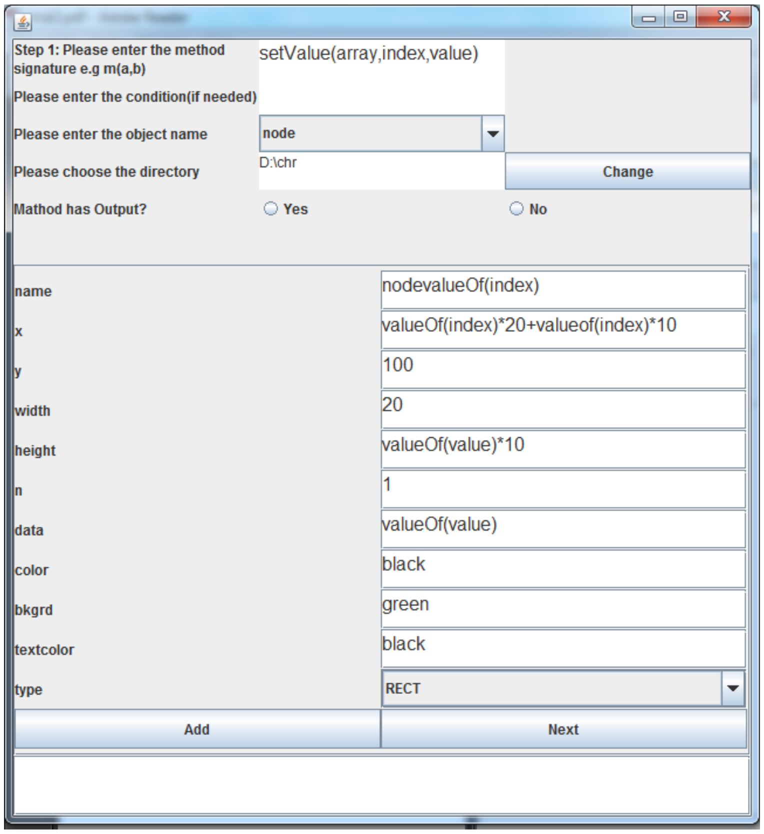
The simpagation rule `idempotence` keeps one copy of the constraint `leq(A,B)`. The rule `reflexivity` removes from the store any constraint of the form `leq(A,A)` since any number is less than or equal to itself. Thus, the constraint is not needed since it does not represent any new piece of information. The simplification rule `antisymmetry` replaces the two constraints `leq(A,B)` and `leq(B,A)` with the built-in constraint `A=B` which is handled by the host language. In the case of the propagation rule `transitivity`, since `A` is less than or equal to `B` and `B` is less than or equal to `C`, the constraint `leq(A,C)` is added to the constraint store to represent the fact that `A` is less than or equal to `C`.

The idea introduced in [12] was to use annotation rules for animation. Users were allowed to specify interesting constraints in a CHR program. Users were able to annotate an interesting constraint with a visual object. That way, whenever such constraint was added to/removed from the constraint store, its associated visual object is also added to/removed from the visual trace. This produces an animation of the execution. In order to have a generic platform, an existing visual scripting tool was used. Such visualization tools offer its users with various objects. For proof of concept, Jawaa[2] [17] was used. It has a group of basic visual objects such as circles, rectangles, etc. It also has a set of actions available for the objects such as changing colors, moving, etc. Using this methodology, it was possible to animate different kinds of algorithms implemented in CHR.

III. Annotation Rules for Java Programs

The aim of this work is to have a generic animation platform for Java programs. The platform should not require any manual modification to the source code. It should also be generic enough to allow for the animation of any type of algorithm. The approach implemented in this work is to extend Java programs with annotation rules to be able to generate animations of their executions without having to do any manual edits. Figure 1 shows the annotation module available to users. It allows the user to link any method

[2]www.cs.duke.edu/csed/jawaa2



Step 1: Please enter the method signature e.g m(a,b)	setValue(array,index,value)
Please enter the condition(if needed)	
Please enter the object name	node
Please choose the directory	D:\chr
Mathod has Output?	○ Yes ○ No
name	nodevalueOf(index)
x	valueOf(index)*20+valueof(index)*10
y	100
width	20
height	valueOf(value)*10
n	1
data	valueOf(value)
color	black
bkgrd	green
textcolor	black
type	RECT

Figure 1. Annotating a Java method

call to a visual object/action. Once the user chooses an object, the window gets automatically populated with its corresponding parameters. The parameters determine how the object looks like. For example, a circle has different parameters including: x-coordinate, y-coordinate, width and color. Annotation rules are dynamic in the sense that users can link one of the parameters to the value of an argument of the method call. This way, different visual objects are dynamically generated without the need of any interaction from the user. The generated annotation rules for Java methods have the following syntax:
method_name(arg$_1, \ldots, arg\_n) \Rightarrow optional\_condition$
$\#par\_1 = val\_1\#par\_2 = val\_2\# \ldots \#par\_n = val\_n$.
Each value in $val_1, val_2, \ldots, val_n$ can be either

1) a constant such as 30, $blue$.
2) $valueOf(arg_i)$ to represent the value of the i^{th} argument of the method call.
3) the keyword $output$ representing the output of the method call.

In the example shown in Figure 1, the method `setValue(array,index,value)` is annotated with a rectangular node. The text inside the node is the same as the `value`. The x-coordinate is a factor of the `index`. The height is a factor of the `value`. The y-coordinate and width are set to constant values. The idea is that every time this method is called an element inside the `array` is set to a new value. This is thus an interesting event for this algorithm. The annotation rule ensures that every time this happens, a rectangular node is added. The location of the node is determined through the index. The node generated for the element at index zero will be placed at a position before the node generated for the element at index one. The rectangular nodes will thus be able to visualize the array as well as any changes to its elements.

IV. Execution Path of the Transformed Java Programs

After the user adds the required annotation rules, the Java program is automatically transformed to a new one. The execution of the new program follows the diagram shown in Figure 2. The Java program is thus firstly parsed. The source-to-source transformer reads the output of the parser and produces a new Java file. In the new program, after any call to an interesting event, a new call is added. This new line of code communicates the invoked interesting event to produce its associated visual object. For instance, in the previous example shown in Figure 1, every time the method `setValue` is called, the visual tracer is notified. The visual tracer adds a new rectangular node. In the case where the

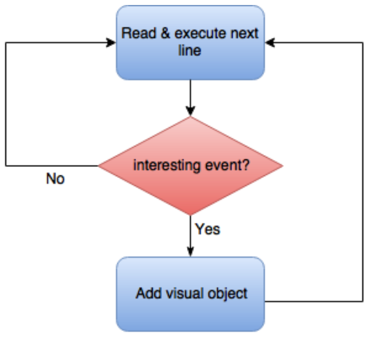


Figure 2. New Execution Path

method has an output value, the visual tracer is also notified with the returned value.

V. Translation to Rules

In order to have a generic platform, the engine of CHRAnimation [12] was maintained. CHRAnimation is able to communicate the constraints added to the constraint store to have a general visual tracer. The tracer utility compares the constraints in the store to the annotated constraints. Once an interesting constraint is detected, its corresponding visual object is modified (added/removed) accordingly. The SWI-Prolog implementation of CHR is used, thus providing portability and ease of use. This aids in having an easy-running animation tool. In the case of annotating Java programs, a CHR program is produced by the source-to-source transformer. To be able to use CHR for activating the visual tracer, the following (abstracted) rule is added for every interesting method *method_name* with k arguments:

```
method_name(arg1,...,argk)==>
communicate(method_name(arg1,...,argk)),
                update_visual_file.
```

For example for the previous example the following rule is added:

```
setValue(A,B,C)==>
            communicate(setValue(A,B,C)),
                       update_visual_file.
```

The propagation rules ensure that every time such an "interesting" constraint is added, the visual tracer is updated with the new information. The visual tracer is automatically able to search through the annotation rules to find any rules associated with the new constraint. Once a corresponding rule is found, the Jawaa animation file is updated with the new visual object. In the case where the interesting method has an output value, the corresponding CHR constraint has one extra argument to account for the returned value.

VI. Example

The following Java code performs the bubble sort algorithm on an array of elements. Using the annotation provided before, the array is visualized at each step. An animation

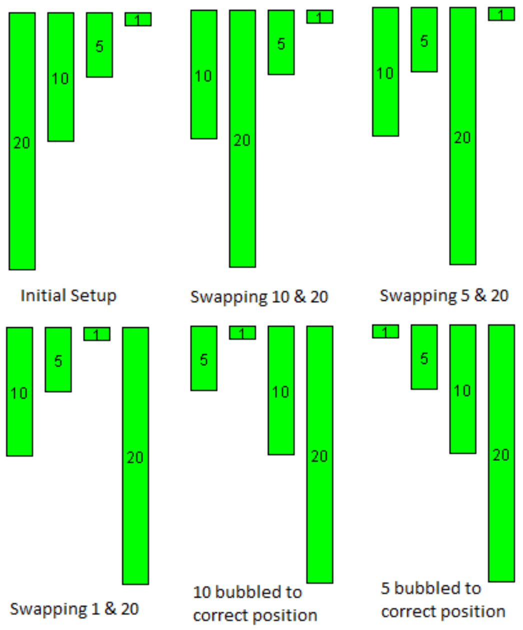


Figure 3. Bubble Sort Animation

of the algorithm is shown in Figure 3. As seen from the figure, the array is visualized through rectangular nodes. Every change performed to the array is visualized. Thus at every step, the new status of the array is shown. The Java program is shown below. [3]

```
public class MySort {
   public static void main(String[]args)
   {
    initializeAndSort();
   }
   public static void setValue(int[]num, int index
       , int newValue)
   {
      num[index]=newValue;
   }

   public static void initializeAndSort()
   {
    int[] numbers = new int[4];
    setValue(numbers, 0, 20);
    setValue(numbers, 1, 10);
    setValue(numbers, 2, 5);
    setValue(numbers, 3, 1);
    boolean swapped = true;
    int temp;
    while (swapped==true) {
      swapped=false;
        for (int i = 0; i < numbers.length - 1; i
           ++) {
          if (numbers[i] > numbers[i + 1])
          {
            temp = numbers[i]; // swap elements
            setValue(numbers,i,numbers[i+1]);
            setValue(numbers, i+1, temp);
            swapped = true;
          }
```

[3]The program uses the same algorithm provided in http://mathbits.com/MathBits/Java/arrays/Bubble.htm.

```
            }
        }
    }
}
```

VII. Conclusions and Future Work

In conclusion, the work presented in the paper provides an animation tool for Java programs. The idea is to utilize annotation rules to determine interesting parts of the code. The tool is able to dynamically annotate the code and generate the required visual objects. The work is one step towards providing a general purpose animation tool through source-to-source transformation. In the future, different parts of Java programs should be annotated such as calls to instantiate new objects through constructors. A prototype of a general purpose rule-based visual language using annotation rules should be also offered to users.

References

[1] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia, "Algorithm animation over the World Wide Web," in *Proceedings of the workshop on Advanced visual interfaces 1996, Gubbio, Italy, May 27-29, 1996*, T. Catarci, M. F. Costabile, S. Levialdi, and G. Santucci, Eds. ACM Press, 1996, pp. 203–212.

[2] S. M. Čisar, R. Pinter, and D. Radosav, "Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3," *International Journal of Computers Communications & Control*, vol. 6, no. 4, pp. 668–680, 2011.

[3] T. Boyle, C. Bradley, P. Chalk, R. Jones, and P. Pickard, "Using Blended Learning to Improve Student Success Rates in Learning to Program," *Journal of Educational Media*, vol. 28, no. 2-3, pp. 165–178, 2003.

[4] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[5] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ System and its Pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003.

[6] M. Kölling, "The Greenfoot Programming Environment," *Trans. Comput. Educ.*, vol. 10, no. 4, pp. 14:1–14:21, Nov. 2010.

[7] W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper, "Mediated transfer: Alice 3 to Java," in *Proceedings of the 43rd ACM technical symposium on Computer science education, SIGCSE 2012, Raleigh, NC, USA, February 29 - March 3, 2012*, L. A. S. King, D. R. Musicant, T. Camp, and P. T. Tymann, Eds. ACM, 2012, pp. 141–146.

[8] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with Jeliot 3," in *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28, 2004*, M. F. Costabile, Ed. ACM Press, 2004, pp. 373–376.

[9] G. Rößling and B. Freisleben, "AnimalScript: An Extensible Scripting Language for Algorithm Animation," in *Proceedings of the 32rd SIGCSE Technical Symposium on Computer Science Education, 2001, Charlotte, North Carolina, USA, 2001*, H. M. Walker, R. A. McCauley, J. L. Gersting, and I. Russell, Eds. ACM, 2001, pp. 70–74.

[10] M. H. Brown and R. Sedgewick, "A System for Algorithm Animation," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, H. Christiansen, Ed. ACM, 1984, pp. 177–186.

[11] M. H. Brown, "Zeus: A System for Algorithm Animation and Multi-View Editing," in *Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan, October 8-11, 1991*, 1991, pp. 4–9.

[12] N. Sharaf, S. Abdennadher, and T. W. Frühwirth, "CHRAnimation: An Animation Tool for Constraint Handling Rules," in *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Proietti and H. Seki, Eds., vol. 8981. Springer, 2014, pp. 92–110.

[13] T. W. Frühwirth, "Theory and Practice of Constraint Handling Rules," *J. Log. Program.*, vol. 37, no. 1-3, pp. 95–138, 1998.

[14] H. Betz, F. Raiser, and T. W. Frühwirth, "A complete and terminating execution model for Constraint Handling Rules," *TPLP*, vol. 10, no. 4-6, pp. 597–610, 2010.

[15] T. W. Frühwirth, "Constraint Handling Rules - What Else?" in *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, ser. Lecture Notes in Computer Science, N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, and D. Roman, Eds., vol. 9202. Springer, 2015, pp. 13–34.

[16] T. W. Frühwirth, *Constraint Handling Rules*, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[17] W. C. Pierson and S. H. Rodger, "Web-based Animation of Data Structures Using JAWAA," in *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, J. Lewis, J. Prey, D. Joyce, and J. Impagliazzo, Eds. ACM, 1998, pp. 267–271.