A Rule-Based Approach for Automatic Interaction Detection and Annotation

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*—The paper introduces an approach that allows for detecting interaction with a graphical display in a rule-based declarative approach. It also introduces the possibility of connecting the interaction to a specific animation step to be performed. The two approaches were embedded into an animation system to aid in teaching algorithms.**

***Index Terms*—Animation, Teaching**

I. Introduction

Over the years, animation systems have developed. There are different animation tools for different programming languages [1], [2]. There are different available animation libraries [3]–[6]. The aim of the work is to allow embedding interactiveness into graphical interfaces in a generic way. Through the proposed approach whenever a user interacts with a graphical display, the engine would be able to automatically detect which object was clicked. The proposed architecture was extended to enable instructors to build up quizzes to aid students in learning different algorithms. As a proof of the concept, the implementation shown in the paper was done with an engine for animating graph and tree algorithms. However, it could be extended to other animation libraries.

The paper is organized as follows: Section II shows how automatic detection of interaction was implemented. Section III discusses in more details the proposed system architecture to be able to annotate click events with animation actions. Section IV shows an example of a quiz built through the proposed system to test depth-first tree-traversal algorithm. We finally conclude with directions to future work.

II. Detecting Clicked Objects Automatically

Once the user clicks anywhere in the graphical display, a Constraint Handling Rules (CHR) [7]–[9] program is queried to determine which object was clicked. The advantage of using CHR is to have easy-to-use declarative rules. In addition, this way, the type of the graphical display used does not matter. Thus, we would be able to embed interaction into graphical displays easily. The nature of CHR allows the program to handle different representations of the same object type. Whenever a new object type needs to be added, the programmer could easily add its corresponding rule. For example, the below snippet shows different representations of a circular object. Whenever a user clicks anywhere, the CHR program is queried with `click(X_clicked,Y_Clicked,Obj)`. `X_clicked` represents the x-coordinate in which the user clicked. `Y_Clicked` represents the y-coordinate. As a result of the query, `Obj` should be bound to the name of the clicked object.

```
object(Name, circle ,[X,Y, Radius ])\click (XC,YC, Obj)
    <=> Value is sqrt((XC - X)^2 + (YC-Y)^2),Value=<
    Radius | clicked(Name, circle ,XC,YC).
object(Name, circle ,[X,Y,W,H])\click (XC,YC, Obj) <=> W
    =H, Value is sqrt((XC - (X+W/2))^2 + (YC-(Y+H/2)
    )^2),Value=<(W/2) |clicked(Name, circle ,XC,YC).
object(Name, oval ,[X,Y,W,H])\click (XC,YC, Obj) <=> W\=
    H, Value is sqrt((XC - (X+W/2))^2 + (YC-(Y+H/2))
    ^2),Value=<(W/2) |clicked(Name, circle ,XC,YC).
```

Each object is represented by a constraint `object(Name,Type,ParList)` where `Name` represents the name/id of the object with type `Type`. `ParList` is a list of parameters of the object. It thus differs according to the object type. Constraint Handling rules CHR [8] has developed over the years into a general purpose language. A CHR program rewrites constraints in the constraint store until a fixed point is reached. At this point no more rules could be applied. A CHR rule is applicable if the constraint store contains constraints matching the head constraints. In addition the pre-condition of the rule (the guard G) has to be satisfied for the rule to be applicable. A CHR program contains user-defined/CHR constraints. It could also deal with built-in constraints handled by the host language. A CHR simpagation rule has the form:
$H_k \setminus H_r \;<=>\; G \mid Body.$
H_k and H_r should only contain CHR constraints. G can only contain built-in constraints. $Body$ could, however, contain CHR and built-in constraints. The constraint store is initialized by the query of the user. On executing such a rule, constraints matching H_r are removed from the constraint store. Constraints matching H_k are kept in the store. The constraints in $Body$ are added to the store. The first rule uses the representation of a circle as an object with the x-coordinate and the y-coordinate of the center in addition to its radius. In this case, in order to make sure that the user clicked inside this circular object, the distance between the clicked coordinates and the center is calculated. The distance has to be less than or equal to the radius to make sure that the

user clicked within the circle. In the second rule, the circular object is represented differently. The parameters `X` and `Y` represent the x and y coordinates of the upper left corner. `W` and `H` represent the width and the height. In case, it is a circle, the width and height have the same value, otherwise it is an oval. The x-coordinate of the center of the object is calculated using the formula `X+W/2`. Its y-coordinate is calculated through `Y+H/2`. A similar calculation to the previous case is also done to make sure that the distance is within the allowed range. This approach thus removes the need of manually adding extra code to the animation libraries to detect interaction.

III. Requirements

This section discusses the requirements that should be embedded to enable using the system. As a proof of concept, it was implemented with an animation engine that generates graphical interfaces for graphs and trees [1]. The rest of the section describes how the system was setup of the system to be able to use it.The rest of the section describes the requirements in more details.

A. Recording the Steps

Different animation libraries such as Jawaa [4] and AnimalScript [3] use the concept of having a script/animation file to produce the required animation. A Jawaa animation file could thus instruct the library for example to add a green rectangle at a certain position and then at a later step change its color to red. Thus, in an animation, graphical objects keep on changing. When a sorting algorithm is visualized, the initial set up would show the unsorted array. As the animation proceeds, the graphical objects keeps getting re-ordered.

The first requirement is that after each change the current state of the animation gets saved in a file whose name reflects the current step in the animation. The first produced file would thus be "objects0.txt", the second would be "objects1.txt" and so on. Each file thus contains the status of the animation at a specific step. Each line in the file contains an object. The structure of the object differs according to its type. For example, the animation result shown in Figure 1 produces the following *objects* file.

Listing 1. Generated Trace File

```
object(2,circle,[a,green,40.0,0.0,50.0,50.0])
object(3,circle,[b,green,0.0,100.0,50.0,50.0])
object(4,circle,[c,green,80.0,100.0,50.0,50.0])
object(5,circle,[d,green,0.0,200.0,50.0,50.0])
object(6,circle,[e,green,0.0,300.0,50.0,50.0])
object(7,circle,[f,green,80.0,200.0,50.0,50.0])
object(8,edge,[2,3])
object(9,edge,[2,4])
object(10,edge,[3,5])
object(11,edge,[3,7])
object(12,edge,[5,6])
```

The line `object(2,circle,[a,green,40.0,0.0,50.0,50.0])`

[1]A paper about this animation system was accepted as a poster in IV2017 as well [10]. More details on the engine are provided in the appendix

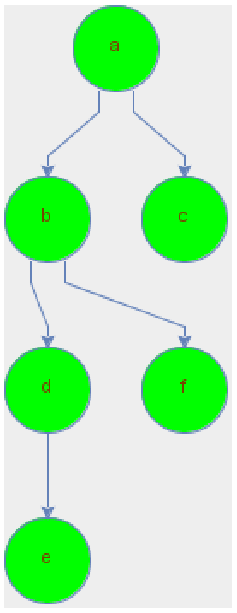


Fig. 1. Visualized Tree

represents a circular object (tree vertex in this case). The object has a name `2`. The list of parameters `[a,green,40.0,0.0,50.0,50.0]` represent the text, color, x-coordinate, y-coordinate width and height correspondingly.

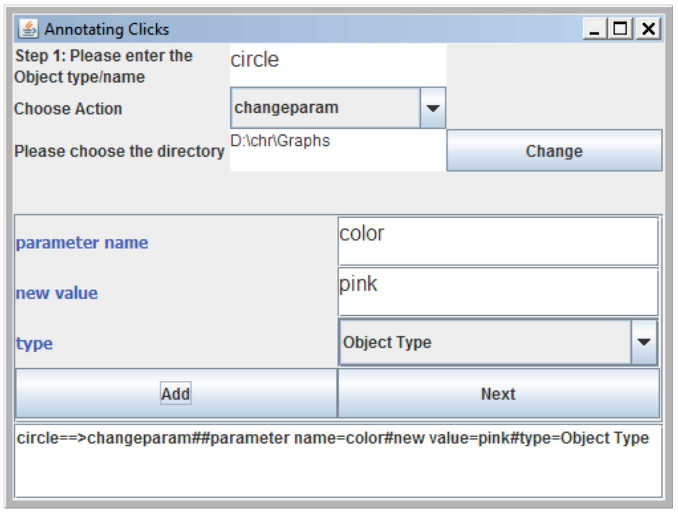
B. Constructing Animation from a File

The animation engine should be able to read a file in the format produced as shown in Section III-A and construct the corresponding animation. For example, the file shown in Listing 1 produces the same tree shown in Figure 1.

C. Annotating Click Events

Users are then able to state what should happen when a certain object is clicked. The system uses the annotation rules introduced in [1], [2]. The proposed idea was to link CHR constraints [1] or Java method calls [2] with graphical objects. That way, whenever constraints are changed or methods are called, their corresponding graphical objects are added to the animation. In [11], the Jawaa animation library was extended with an action that detects whether a certain object was clicked. Whenever such object was clicked the user can instruct the library to do a specific action. Detecting where the user clicked had to, however, be done through extending the code of the animation library. The idea is to enable having this kind of association for different animations and animation libraries. The approach introduced in Section II enables detecting interaction. It is also able to automatically detect which object was clicked and its type.

Through the system, users are able to provide annotations for click events as shown in Figure 2. The user should provide the name of the object or the type of the object that when clicked should trigger an animation effect. For example in Figure 2, the effect is `changeparam` (similar to the same


Annotating Clicks

Step 1: Please enter the Object type/name: circle

Choose Action: changeparam

Please choose the directory: D:\chr\Graphs

parameter name: color

new value: pink

type: Object Type

```
circle==>changeparam##parameter name=color#new value=pink#type=Object Type
```

Fig. 2. Annotating Clicks

command provided in Jawaa). This action triggers a change in one of the parameters of the object. Thus, the user should also provide the name of the parameter that should change and its new value. In the example shown in Figure 2, the annotation is triggered each time an object of type `circle` is clicked. The annotation could be linked to an object with a specific name as well. This produces the following annotation rule: `circle==>changeparam#parameter name=color #new value=pink#type=Object Type`

For example in the case where the user clicks on node "a" in the tree shown in Figure 1, three things happens:

1) The animation engine communicates the clicked coordinates to the click detection engine. The click detection engine queries the CHR program to find out which object was clicked and its type. In this case it detects that the object named `2` is clicked and the type of it is circle.
2) At this point the click annotations' engine is activated. It compares the click annotations inserted by the user to check whether any of the annotations is applicable at this point. In this example, the annotation rule is applicable since the clicked object has the type: circle.
3) The click annotations' engine notifies the animation engine of the action that should be performed. In this example, the color of the node should change to `pink`. The result in this example is shown in Figure 3 where the color of the node has changed to `pink`.

The proposed architecture of the system is shown in Figure 4.

IV. Using Click Annotations For Generating Quizzes

The previously proposed architecture could be used to build up quizzes to test students in different algorithms. In this case, the system should be extended with following features:

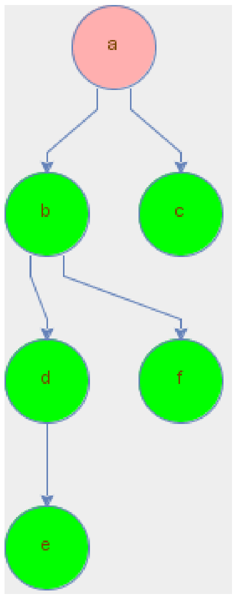


Fig. 3. Result of Clicking on node a

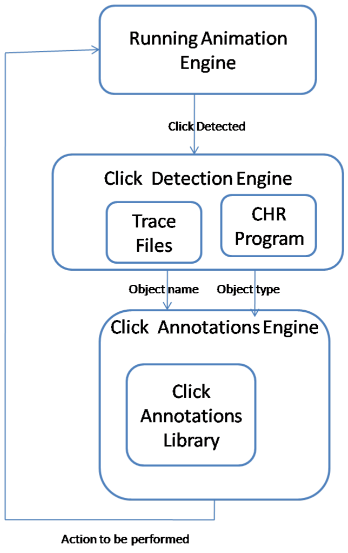
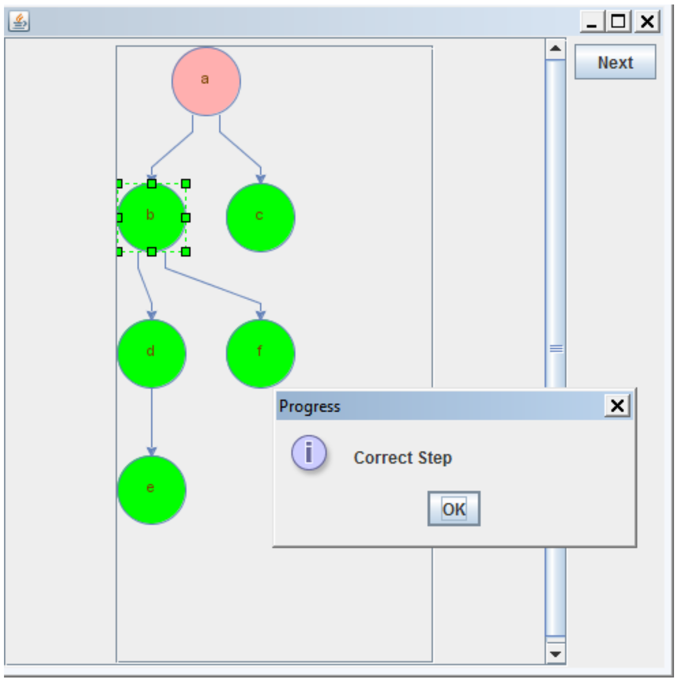
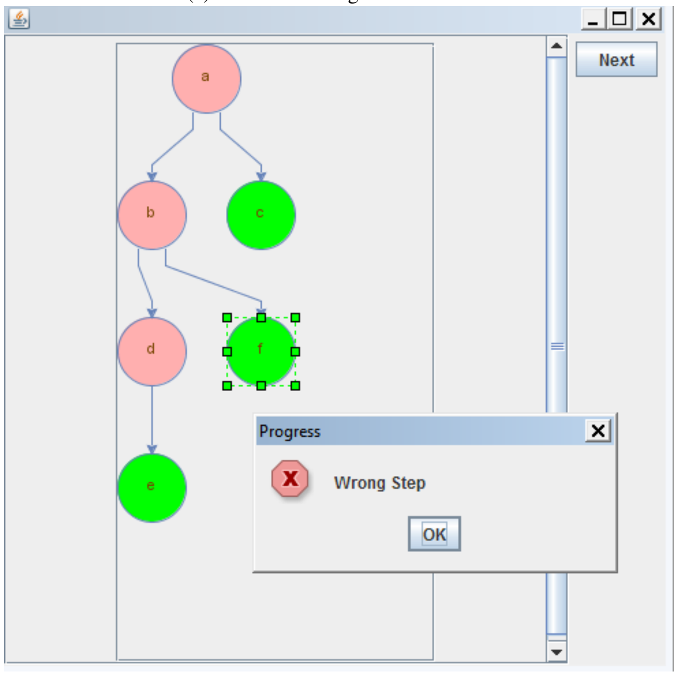


Fig. 4. Proposed Architecture

1) The instructor should first run the complete animation to produce all the trace files discussed in Section III-A.
2) The system should allow the instructor to set a threshold for the start of the animation. For example, in case the instructor sets the threshold to `10`, then the first screen the student sees is the result of building up the animation using the file "objects10.txt" as discussed in Section III-B.
3) The click actions should be annotated by the instructor.
4) Each time the student clicks an object, the generated graphical view is compared with the previously saved trace file from step 1. If they are equivalent, then the student took the correct step. Otherwise, the student made a mistake and the action is rolled back. The student



(a) Student clicking a correct node



(b) Student clicking a wrong node

Fig. 5. Testing the depth-first tree traversal algorithm

is notified with the progress at each step.

An example for using the system is shown in Figure 5. In this example, the depth-first traversal algorithm is being tested. After clicking on node a the student clicked on node b which is a correct step as shown in Figure IV. Afterwards the student clicked on node d and then tried to click on node f which is incorrect. A corresponding message thus appears.

V. Conclusions & Future Work

The paper presented a framework that is able to detect interaction automatically. Interaction was used to build testing quizzes for students for different algorithms. The annotation of click events was also introduced. As a proof of concept, the interaction detection and quizzes building utility was tested with a Java-based library for animating graph and tree algorithms. However, the methodology could be extended to other animation libraries. The annotation of sequences of clicks with specific properties should be also added to the system.

References

[1] 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.*, ser. Lecture Notes in Computer Science, M. Proietti and H. Seki, Eds., vol. 8981. Springer, 2014, pp. 92–110.

[2] ——, "A Rule-Based Approach for Animating Java Algorithms," in *20th International Conference Information Visualisation, IV 2016, Lisbon, Portugal, July 19-22, 2016*, E. Banissi, M. W. M. Bannatyne, F. Bouali, R. Burkhard, J. Counsell, U. Cvek, M. J. Eppler, G. G. Grinstein, W. Huang, S. Kernbach, C. Lin, F. Lin, F. T. Marchese, C. M. Pun, M. Sarfraz, M. Trutschl, A. Ursyn, G. Venturini, T. G. Wyeld, and J. J. Zhang, Eds. IEEE Computer Society, 2016, pp. 141–145. [Online]. Available: http://dx.doi.org/10.1109/IV.2016.55

[3] 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. [Online]. Available: http://doi.acm.org/10.1145/364447.364541

[4] 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.

[5] J. Stasko, "Animating algorithms with xtango," *SIGACT News*, vol. 23, no. 2, pp. 67–71, May 1992. [Online]. Available: http://doi.acm.org/10.1145/130956.130959

[6] S. Abdennadher and N. Sharaf, "Visualization of Constraint Handling Rules through Source-to-Source Transformation," in *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, Budapest, Hungary*, 2012.

[7] T. W. Frühwirth, "Theory and practice of constraint handling rules," *J. Log. Program.*, vol. 37, no. 1-3, pp. 95–138, 1998. [Online]. Available: http://dx.doi.org/10.1016/S0743-1066(98)10005-5

[8] ——, *Constraint Handling Rules*. Cambridge University Press, 2009.

[9] ——, "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. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21542-6_2

[10] N. Sharaf, S. Abdennadher, and T. Frühwirth, "CHR-Graph: A Platform for Animating Tree and Graph Algorithms," in *21st International Conference Information Visualisation, IV 2017, London, UK, July 11-14, 2017*.

[11] N. Sharaf, S. Abdennadher, T. Frühwirth, and D. Gall, "Animating cognitive models and architectures: A rule-based approach," in *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, ser. EPiC Series in Computing, vol. 41. EasyChair, 2016, pp. 253–265.

Appendix

The animation library used throughout the paper is currently under consideration for publication. It allows users to build graphs and trees by using animation files similar to the architecture provided in Jawaa [4]. The library provides the following commands:

1) `node n t c` which adds a new node with the name `n`, text `t` and color `c`.
2) `edge s t l` which ads a new edge from the node named `s` to the node named `t`. The label (which is an optional parameter) is `l`.
3) `updateNode n par val` which updated the parameter `par` of node `n` to have the new value `val`. The parameter could thus be the text of the node or its color for example.

For example the tree shown in Figure 1 is built using the following animation file:

```
node 2 a green
node 3 b green
node 4 c green
node 5 d green
node 6 e green
node 7 f green
edge 2 3
edge 2 4
edge 3 5
edge 3 7
edge 5 6
```