Visualization of Constraint Handling Rules

Nada Sharaf[1], Slim Abdennadher[1] and Thom Frühwirth[2]
[1] *The German University in Cairo, Egypt;* [2] *Ulm University, Germany*
(e-mail:{nada.hamed, slim.abdennadher}@guc.edu.eg, thom.fruehwirth@uni-ulm.de)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Constraint Handling Rules (CHR) has matured into a general purpose language over the past two decades. Any general purpose language requires its own development tools. Visualization tools, in particular, facilitate many tasks for programmers as well as beginners to the language. The article presents on-going work towards the visualization of CHR programs. The process is done through source-to-source transformation. It aims towards reaching a generic transformer to visualize different algorithms implemented in CHR.
Note: An extended abstract / full version of a paper accepted to be presented at the Doctoral Consortium of the 30th International Conference on Logic Programming (ICLP 2014), July 19-22, Vienna, Austria.

KEYWORDS: Constraint Handling Rules, Algorithm Visualization, Source-to-Source Transformation

---

1 Introduction

Although Constraint Handling Rules (CHR)(Frühwirth 1998) was introduced as a language for writing constraint solvers, it has developed into a general purpose language over the years. CHR is a committed choice language. A CHR program consists of multi-headed guarded rules. In CHR, predicates are transformed into simpler ones until they are solved. CHR has a number of implementations. However, the most prominent ones are embedded in Prolog.

Since CHR has developed into a general purpose language, CHR programmers can now write programs to implement general algorithms such as sorting algorithms, graph algorithms, etc. With such algorithms, programs could get very long and in some cases complicated. Thus development tools and especially tracing and visualization tools are very useful and sometimes even necessary. As discussed in (Hundhausen et al. 2002), algorithm visualization technologies are useful in many cases such as in practical laboratories, for in-class discussions, or in assignments where students could for example produce their own visualizations. It could help instructors find bugs quickly. Moreover, such visualizations could be useful for debugging and tracing the implementations of different algorithms.

There are different methods for embedding visualization features into a CHR program. One of them is to alter the compiler or the CHR runtime system. This solution is however not recommended since performing such changes is not an easy task for any programmer especially a beginner. Thus the adopted approach is to use source-to-source transformation to eliminate the need of doing any changes to the running system.

Program transformation or source-to-source transformation, allows developers to add or change the behavior of programs without manually modifying the initial code. In addition, according to (Loveman 1977), source-to-source transformation could be useful for improving the performance of different programs

In (Abdennadher and Sharaf 2012), a first attempt towards visualizing CHR programs was presented. The presented tool was able to visualize the execution of CHR programs was realized through source-to-source transformation. In addition, it was able to visualize CHR constraints as objects. The system, however, lacked generality and required ad-hoc hard-wired inputs. In addition visualizing algorithms implemented through the different CHR programs was not possible. Some systems (Schulte 1997; Simonis and Aggoun 2000) provided visualization options for constraint programs. However, the focus was on the search space and trees rather than the executed algorithms. This paper thus presents a new approach that aims at visualizing CHR solvers in a generic way overcoming the problems of the old tool. Through the new system, various algorithms implemented through CHR could be visualized.

The final goal of the project is to have a general source-to-source transformation tool that is able to automatically add different extensions to CHR solvers.

The paper is organized as follows: Section 2 introduces CHR through an example. Section 3 discusses in more details the suggested architecture of the system. An example of the output of the system is shown in Section 4. Finally, some conclusions and directions for future work are shown in Section 5.

2 Constraint Handling Rules

This section presents an example of a CHR program to introduce the syntax and semantics of CHR (Frühwirth 2009). In CHR, two types of constraints are available. The first type is the built-in constraints that are provided through the host language. The second type of constraints is the CHR or user-defined constraints that are defined through the rules of the program A CHR program consists of *simpagation rules* with the form:

$$name \; @ \; H_k \setminus H_r \;\Leftrightarrow\; G \mid B.$$

The name of the rule precedes the @ sign and is optional. The head of the CHR rule, comes before the ($\Leftrightarrow$). It should only contain a conjunction of CHR constraints. As seen from the previous rule, there are two parts in the head namely H_k and H_r. H_k contains the constraints that are kept after the rule is executed. However, the constraints in H_r are removed after executing the rule. The guard (G) is optional and should only contain built-in constraints. Finally, the body (B) could contain both CHR and built-in constraints. The constraints in the body are added to the constraint store on executing the rule.

A rule is executed only if the head constraints match some of the constraints in the constraint store (Frühwirth 2008). In addition, the guard has to be satisfied for the rule to be executed. At the beginning of the execution, the constraint store is empty. It is initialized by the constraints in the query.

There are special cases of simpagation rules which are simplification and propagation rules. Whenever H_k is empty, the resulting rule is a “simplification” rule. The head of a simplification rule contains CHR constraints that are removed once the rule is executed. Thus through *simplification* rules, CHR constraints are replaced by simpler ones. Thus the format of a simplification rule is:

$$H_r \quad \Leftrightarrow \quad G \quad | \quad B.$$

In *propagation rules*, H_r is empty. Consequently, the rule does not remove any constraints from the store. It only adds the constraints in the body to the store. Propagation rules have the following format:

$$H_k \quad \Rightarrow \quad G \quad | \quad B.$$

The following example is for a CHR program that sorts numbers. The numbers are fed into the solver using the constraint `list`. For example, the constraint `list(I,V)` means that the cell at index `I` of the list has the value `V`. The solver contains the simplification rule `sortlist`.

```
sortlist @ list(Index1,V1), list(Index2,V2) <=> Index1<Index2 , V1>V2 |
                           list(Index2,V1), list(Index1,V2).
```

The rule `sortlist` makes sure that a number precedes another one in the list if and only if it is indeed smaller than it. If this is not the case, the two numbers are swapped. Consequently, applying such a rule results in a sorted sequence of numbers. For example if the input to the solver is `list(0,7), list(1,6), list(2,4)`, execution proceeds as follows:

$$\begin{array}{c}
\underline{\mathit{list}(0,7), \mathit{list}(1,6)}, \mathit{list}(2,4) \\
\Downarrow \\
\mathit{list}(1,7), \underline{\mathit{list}(0,6), \mathit{list}(2,4)} \\
\Downarrow \\
\underline{\mathit{list}(1,7), \mathit{list}(2,6)}, \mathit{list}(0,4) \\
\Downarrow \\
\mathit{list}(2,7), \mathit{list}(1,6), \mathit{list}(0,4)
\end{array}$$

As seen from the previous execution sample, every time a new number is added to the sequence or the store, the program makes sure it is placed in the correct position with respect to the already existing elements. Accordingly, after all numbers are added, the resulting sequence is a sorted one. The semantics implemented in SWI Prolog is the refined operational semantics (Duck et al. 2004). It makes sure that constraints are processed from the left to the right and that rules are executed in a top-bottom approach as demonstrated through the previous example. For example after `list(2,4)` is added to the constraint store, the rule `sortlist` is executed since `6` and `4` are not sorted in the sequence. Thus, `list(0,6)` and `list(2,4)` are removed from the constraint store. They should be replaced by `list(2,6)` and `list(0,4)`. However, these two constraints are added to the store one by one. Once `list(2,6)` is added to the store and even before adding `list(0,4)`, the rule `sortlist` is executed using the two constraints `list(2,6)`, `list(1,7)`. A more detailed view of the execution steps is:

An Empty Constraint Store	
$\Downarrow$	Adding `list(0,7)` to the store
$\mathit{list}(0,7)$	
$\Downarrow$	Adding `list(1,6)` to the store
$\underline{\mathit{list}(1,6),\mathit{list}(0,7)}$	
$\Downarrow$	`sortlist` removing `list(0,7)`, `list(1,6)` and adding adding `list(1,7)` as a first step
$\mathit{list}(1,7)$	
$\Downarrow$	Adding `list(0,6)` to the store
$\mathit{list}(0,6),\mathit{list}(1,7)$	
$\Downarrow$	Adding `list(2,4)` to the store
$\underline{\mathit{list}(2,4),\mathit{list}(0,6)},\mathit{list}(1,7)$	
$\Downarrow$	`sortlist` removing `list(2,4), list(0,6)` and adding adding `list(2,6)` as a first step
$\underline{\mathit{list}(2,6),\mathit{list}(1,7)}$	
$\Downarrow$	`sortlist` removing `list(2,6), list(1,7)` and adding adding `list(2,7)` as a first step
$\mathit{list}(2,7)$	
$\Downarrow$	Adding `list(1,6)` to the store
$\mathit{list}(1,6),\mathit{list}(2,7)$	
$\Downarrow$	Adding `list(0,4)` to the store
$\mathit{list}(0,4),\mathit{list}(1,6),\mathit{list}(2,7)$	

3 System Architecture

This section introduces the adopted architecture and transformation approach. The tool presented by (Abdennadher and Sharaf 2012) was able to add visualization features to CHR programs. The tool however lacked the possibility of visualizing different algorithms implemented through `CHR`. In order to extend the tool, the user was always required to enter specific hard-wired inputs. The focus is now for a new and a more general approach. As shown in Figure 1, the general architecture of the workbench consists of several modules. At the beginning, the `CHR` program is fed into the parser. In addition to parsing the input file, the parser also extracts the needed information and represents it in the required format for the transformer.

The transformer uses the "relational normal form" presented in (Frühwirth and Holzbaur 2003). This form uses some special `CHR` constraints to encode the different constituents of a `CHR` rule. Such constraints include `head`, `body` and `guard`. The parser thus represents the information about the different rules using the specified form. For example, `head(sortlist,'list(Index1,V1)',remove)` represents the fact that the rule `sortlist` has the constraint `list(Index1,V1)` in its head. In addition, this constraint is removed on executing the rule since it is a simplification rule. Finally, the new solver is generated by the transformer. Unlike the form presented in (Frühwirth and Holzbaur 2003), the transformer neglects the identifier of the head constraint since it is not needed.

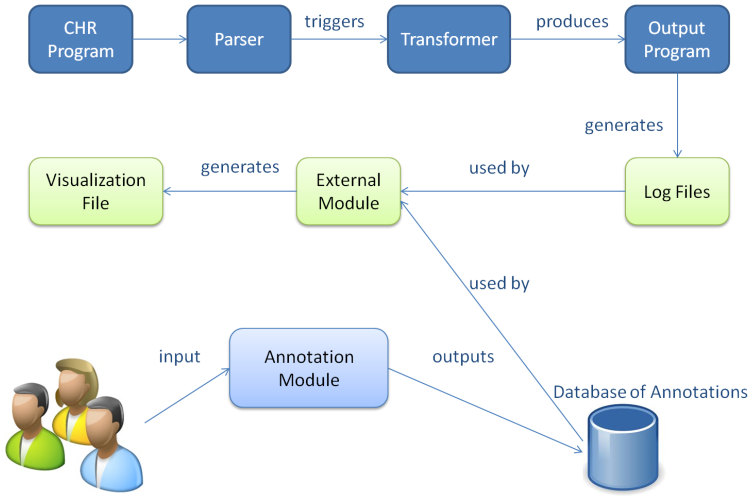


Fig. 1: The General Architecture

In addition, the system makes use of a new module called the "Annotation Module" explained in more detail in Section 3.1.

The output programs are normal CHR programs that could run with SWI-Prolog. Generally, when the output solver is running, log files are generated as shown in Figure 1. These files should contain information regarding the executed rules that could then be used by an optional external module.

The external module is utilized by the visualization extension. This module reads the log files generated by the new program. In addition it makes use of the output of the *Annotation Module* in order to produce the visualization file needed by the visual tracer. This visual tracer could be any visualization program. For proof of concept we used Jawaa (Pierson and Rodger 1998).

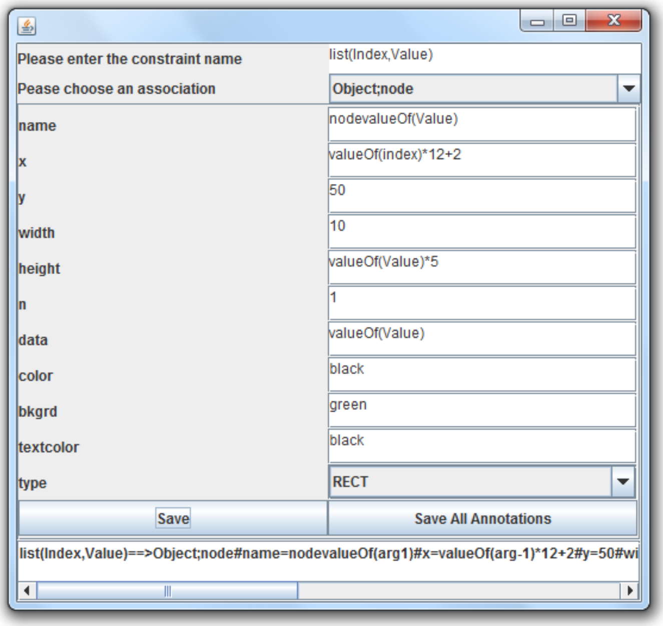
3.1 Annotation Module

This module was proposed as a step towards eliminating the need of entering algorithm-specific information. The new approach this paper introduces also aimed at a more general visual tracer that does not need to be changed every time the algorithm differs. The idea is to have a more basic visualization tool and a more intelligent transformation process. Thus, we decided to out-source the actual visualization process to existing tools such as Jawaa, OpenSCAD[1], etc. Such systems provide different sets of visualization objects and possibly actions as well. Nevertheless, the need of connecting the transformed CHR programs to these systems in a generic way to be able to visualize any algorithm remained. As introduced in (Kerren and Stasko 2002), algorithm animation or software visualization produces abstractions for the data and the operations of an algorithm. The different states of the algorithm are represented as images that are animated according to the different interactions between such states.

---

[1] `http://www.openscad.org/`

The adapted idea in the new system is to visualize the CHR constraints themselves as objects. After transforming the initial program, the rules of the new program modify the constraints and thus the objects. Consequently, the execution of each rule adds one step to the visual tracer. Viewing the sequence of objects thus produces an animation showing how the rules acted on the constraints (or objects) and thus visualizes the algorithm. Accordingly, the "*Annotation Module*" was suggested. The module extends the system with the annotation functionality which allows it to deal with different visualization tools while keeping a general scheme. When deploying the system, users enter the annotations or the mappings between the different constraints and the objects provided through the visualization system. Figure 2 shows how the user mapped the constraint `list/2` to the visual object "Node" that Jawaa offers. In addition to stating the name of the object, the user also states how the arguments of the constraints affect the parameters needed for the object. The parameters for the *Node* object include the name, the x-coordinate, the y-coordinate, the width, the height in addition to the text. The values the user enters for these parameters are also associated with the annotations. The values could use some of the arguments of the constraints. As shown in Figure 2, the constraint is mapped into a *Node* object which means that everytime a `list/2` constraint is added to the store a new *Node* is visualized. The name of the *Node* is chosen to be *nodevalueOf(Value)* which means that the each element in the list will have a node with a name corresponding to its value. For example, the Node corresponding to the number `7` is a `node7`. The x-coordinate is calculated with the following formula $valueOf(Index) * 12 + 2$. Through this formula the Node of the element at index `0` is placed at the x-coordinate `2`. The Node of the second element (with index `1`) is placed at the x-coordinate `14`. The third element



Please enter the constraint name	list(Index,Value)
Pease choose an association	Object;node
name	nodevalueOf(Value)
x	valueOf(index)*12+2
y	50
width	10
height	valueOf(Value)*5
n	1
data	valueOf(Value)
color	black
bkgrd	green
textcolor	black
type	RECT

```
list(Index,Value)==>Object;node#name=nodevalueOf(arg1)#x=valueOf(arg-1)*12+2#y=50#wi
```

Fig. 2: Annotating CHR constraints.

is placed at `26`. Given that the width is determined to be 10, this means that there is a 2 pixels gap between any two consecutive nodes. The y-coordinate is determined to be `50` for all of the Nodes. The height, on the other hand, is five times the value of the element represented through `valueOf(Value)*5`. The node contains only one piece of information which is the value of the element. The outline color of the node is black and the background color is green.

Whenever the user saves the annotation, the following XML file is produced keeping track of all of the associations to the constraint.

```
<association>
<constraint name="list(Index,Value)">
<add name="node" parameters="name=nodevalueOf(arg1)#x=valueOf(arg0)
    *12+2#y=50#
width=10#height=valueOf(arg1)*5#n=1#data=valuef(Value)#color=black#
    bkgrd=green#textcolor=black#type=RECT" type="arg1"/>
</constraint>
</association>
```

3.2 Transformation Module

This section shows how the rules in the original `CHR` program are transformed to be able to interact with the visualization system. Source-to-source transformation was used in order to avoid any manual changes. The basic scheme is similar to the one shown in (Abdennadher and Sharaf 2012). In more details, the new solver interacts with an external module. This module was implemented in Java. The external module uses the information propagated through the solver in addition to the saved annotations to produce a file that could be animated through the visualization system such as Jawaa or OpenSCAD. To have a step-by-step animation, the execution of each rule communicates the needed information.

In general any `CHR` rule of the form:

$$\mathit{rule} \ @ \ H_k \setminus H_r \ <=> \ G \mid \mathit{Body}.$$

is transformed to a rule of the form:

$$\mathit{rule} \ @ \ H_k \setminus H_r \ <=> \ G \mid \mathit{communicate\_hk}(H_k),\ \mathit{communicate\_hr}(H_r),\ \mathit{Body}.$$

As seen from the scheme, the functionality of the rule is kept intact. However, the auxiliary predicate `communicate/1` is used within the new body of the rule. The aim of this predicate is to send the head of the rule to the external module. This module can then start to act upon the information to add the next visualization step.

Actually, only the heads that are removed from the constraint store can affect the visualization. In such a case, the visualized objects for these constraints should be removed from the visualization window.

In some cases, the constraints of the head will not affect the visualization. The transformer can then be instructed to produce a new solver that does not communicate to the external module the head constraints. In such output solvers the old rules are kept intact:

$$\mathit{rule} \ @ \ H_k \setminus H_r \ <=> \ G \mid \mathit{Body}.$$

However, such transformation is not sufficient since the constraints in the body of the rule were not communicated. The body-constraints are responsible for adding new constraints/objects to the trace. Since with the proposed system each constraint maps to an object, a generic way of solving this problem is to add for each constraint `cons/n` a rule with the form:

$$cons(arg_1,\ arg_2,\ arg_3,\ \dots\ , arg_n) \Rightarrow communicate(cons(arg_1,\ arg_2,\ arg_3,\ \dots\ , arg_n)).$$

The previous rule is a propagation rule which does not affect the constraint store. It is however triggered once a constraint of the form $cons(arg_1,\ arg_2,\ arg_3,\ \dots\ , arg_n)$ is added to the store communicating to the external module the new constraint in the store. If this constraint has a corresponding annotation, the module adds the needed visualization step. In addition, if a rule adds multiple constraints that have corresponding annotations, they will be animated one by one. Thus the visualization step is done once the constraint is added to the store. Such propagation rules are added at the beginning of the program to make sure they are executed once a constraint is added and before executing any other applicable rule.

4 Jawaa Example

This section shows how the sorting algorithm shown in Section 2 is visualized using Jawaa. The output solver has the following extra rule at its beginning:

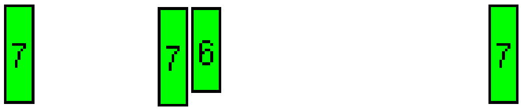
```
list(V0,V1) ==> communicate(list(V0,V1)).
```

In addition the rule `sortlist` is modified such that it communicates to the external module its head constraints since their corresponding objects need to be removed. Thus the new rule has the following format:

```
sortlist @ list(Index1,V1), list(Index2,V2) <=> Index1<Index2 , V1>V2 |
        communicate_hr(list(Index1,V1)), communicate_hr(list(Index2,V2)),
        list(Index2,V1), list(Index1,V2).
```

To have an animation through Jawaa, a “anim” file that contains all of the animation details needs to be used. The external module makes use of the information communicated from the new solver in addition to the saved annotations in order to generate the corresponding animation file. The external module is able to dynamically build up the animation file step by step.

For the query `list(7,0), list(6,1),list(4,2)`, the generated animation file after processing the query is shown in Appendix A. As seen from the file, each time a new `list` constraint was generated, the corresponding node was added. The first generated constraint `list(0,7)` adds to the file:
`node node7 2 50 10 35 1 7 black green black RECT` which adds a node with x-coordinate: `2`, y-coordinate: `50`, width: `10`, height `35`. The text in the node is `7`. The execution of the solver keeps on adding `list` constraints and thus Jawaa nodes are added. In addition, everytime two elements are swapped, their corresponding nodes are removed since the rule `sortlist` is a simplification rule. The resulting animation is shown in Figure 3.

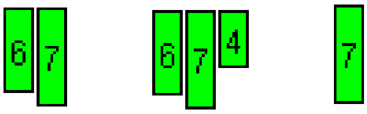


(a) adding list(0,7)

(b) adding list(1,6)

(c) removing list(0,7) list(1,6)

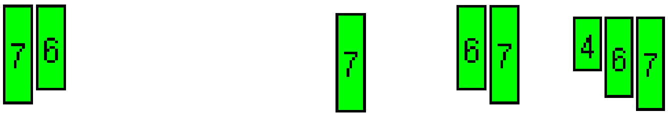
(d) adding list(1,7)



(e) adding list(0,6)

(f) adding list(2,4)

(g) removing list(0,6) list(2,4)



(h) adding list(2,6)

(i) removing list(2,6) list(1,7)

(j) adding list(2,7)

(k) adding list(1,6)

(l) adding list(0,4)

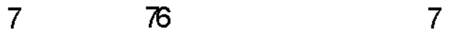
Fig. 3: Sorting the sequence 7, 6, 4 using the sorting solver introduced in Section 2.

```
<association>
<constraint name="list(Index,Value)">
<add name="text" parameters="name=nodevalueOf(arg1)#x=valueOf(arg0)
    *12+2#y=50#text=valueOf(arg1)#color=black#size=30" type="Object"/
    >
</constraint>
</association>
```

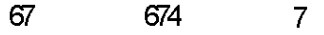
Figure 4 shows the resulting animation using the query `list(0,7), list(1,6), list(0,4)`.

5 Conclusion

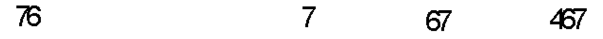
The paper introduced a transformation approach that is able to add visualization features to CHR solvers. The new system overcomes the drawbacks of the old approach. As seen, the need for explicit inputs is removed. The system was able to map constraints into generic existing objects. There was also no need to change the compiler or the runtime system. We are currently in the process of producing a portable version of the application. It will provide users with running examples in addition to patterns for the annotation module. Throughout the paper, Jawaa was used for proof of concept. However, for the future, the system will also be tested with different tools such as OpenSCAD. This approach can be used with an existing visualization system since the intelligence is moved to the transformer and not the tracer. The possibility of using the system to visualize



(a) adding list(0,7)
(b) adding list(1,6)
(c) removing list(0,7) list(1,6)
(d) adding list(1,7)



(e) adding list(0,6)
(f) adding list(2,4)
(g) removing list(0,6) list(2,4)



(h) adding list(2,6)
(i) removing list(2,6) list(1,7)
(j) adding list(2,7)
(k) adding list(1,6)
(l) adding list(0,4)

Fig. 4: Sorting the sequence 7, 6, 4 using the sorting solver introduced in Section 2.

different CHR semantics should be examined. The final goal is to have a generic transformation workbench for CHR. The possibility of having data about the annotations should be also studied. In such a case, the combination of constraints could be used to fire an event.

References

Abdennadher, S. and Sharaf, N. 2012. Visualization of CHR through Source-to-Source Transformation. In *ICLP (Technical Communications)*, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 109–118.

Duck, G. J., Stuckey, P. J., García de la Banda, M. J., and Holzbaur, C. 2004. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 90–104.

Frühwirth, T. 1998. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming 37,* 1-3 (October), 95–138.

Frühwirth, T. 2008. Welcome to Constraint Handling Rules. 1–15.

Frühwirth, T. 2009. *Constraint Handling Rules.* Cambridge University Press.

Frühwirth, T. and Holzbaur, C. 2003. Source-to-Source Transformation for a Class of Expressive Rules. In *APPIA-GULP-PRODE*, F. Buccafurri, Ed. 386–397.

Hundhausen, C., Douglas, S., and Stasko, J. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing 13,* 3, 259–290.

Kerren, A. and Stasko, J. 2002. Chapter 1 Algorithm Animation. In *Software Visualization*, S. Diehl, Ed. Lecture Notes in Computer Science, vol. 2269. Springer Berlin / Heidelberg, 1–15.

Loveman, D. B. 1977. Program Improvement by Source-to-Source Transformation. *J. ACM 24*, 121–145.

Pierson, W. C. and Rodger, S. H. 1998. Web-based animation of data structures using JAWAA. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education.* SIGCSE '98. ACM, New York, NY, USA, 267–271.

Schulte, C. 1997. Oz Explorer: A Visual Constraint Programming Tool. In *ICLP*, L. Naish, Ed. MIT Press, 286–300.

Simonis, H. and Aggoun, A. 2000. Search-Tree Visualisation. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo, and J. Maluszynski, Eds. Lecture Notes in Computer Science, vol. 1870. Springer, 191–208.

Appendix A Jawaa Animation File

```
delay 2500
begin
node node7 2 50 10 35 1 7 black green black RECT
end
delay 2500
begin
node node6 14 50 10 30 1 6 black green black RECT
end
delay 2500
begin
remove node7
remove node6
end
delay 2500
begin
node node7 14 50 10 35 1 7 black green black RECT
end
delay 2500
begin
node node6 2 50 10 30 1 6 black green black RECT
end
delay 2500
begin
node node4 26 50 10 20 1 4 black green black RECT
end
delay 2500
begin
remove node6
remove node4
end
delay 2500
begin
node node6 26 50 10 30 1 6 black green black RECT
end
delay 2500
begin
remove node7
remove node6
end
delay 2500
begin
node node7 26 50 10 35 1 7 black green black RECT
end
delay 2500
```

```
begin
node node6 14 50 10 30 1 6 black green black RECT
end
delay 2500
begin
node node4 2 50 10 20 1 4 black green black RECT
end
```

Appendix B Jawaa Animation File using Text Annotation Object

```
delay 2500
begin
text node7 2 50 7 black 30
end
delay 2500
begin
text node6 14 50 6 black 30
end
delay 2500
begin
remove node7
remove node6
end
begin
text node7 14 50 7 black 30
end
delay 2500
begin
text node6 2 50 6 black 30
end
delay 2500
begin
text node4 26 50 4 black 30
end
delay 2500
begin
remove node6
remove node4
end
begin
text node6 26 50 6 black 30
end
delay 2500
begin
remove node7
remove node6
```

```
end
begin
text node7 26 50 7 black 30
end
delay 2500
begin
text node6 14 50 6 black 30
end
delay 2500
begin
remove node4
end
begin
text node4 2 50 4 black 30
end
```