CHRAnimation: An Animation Tool for Constraint Handling Rules

Nada Sharaf[1(✉)], Slim Abdennadher[1], and Thom Frühwirth[2]

[1] The German University in Cairo, Cairo, Egypt
[2] Ulm University, Ulm, Germany
{nada.hamed,slim.abdennadher}@guc.edu.eg,
thom.fruehwirth@uni-ulm.de

Abstract. Visualization tools of different languages offer its users with a needed set of features allowing them to animate how programs of such languages work. Constraint Handling Rules (CHR) is currently used as a general purpose language. This results in having complex programs with CHR. Nevertheless, CHR is still lacking on visualization tools. With Constraint Handling Rules (CHR) being a high-level rule-based language, animating CHR programs through animation tools demonstrates the power of the language. Such tools are useful for beginners to the language as well as programmers of sophisticated algorithms. This paper continues upon the efforts made to have a generic visualization platform for CHR using source-to-source transformation. It also provides a new visualization feature that enables viewing all the possible solutions of a CHR program instead of the don't care nondeterminism used in most CHR implementations.

Keywords: Constraint Handling Rules · Algorithm visualization · Algorithm animation · Source-to-source transformation

1 Introduction

Constraint Handling Rules (CHR) [1] is a committed-choice rule-based language with multi-headed rules. It rewrites constraints until they are solved. CHR has developed from a language for writing constraint solvers into a general purpose language. Different types of algorithms are currently implemented using CHR.

So far, visually tracing the different algorithms implemented in CHR was not possible. Such visual tools are important for any programming language. The lack of such tools makes it harder for programmers to trace complex algorithms that could be implemented with CHR. Although the tool provided through [2] was able to add some visualization features to CHR, it lacked generality. It was only able to visualize the execution of the different rules in a step-by-step manner. In addition to that, it was able to visualize CHR constraints as objects. However, the choice of the objects was limited and the specification of the parameters of the different objects was very rigid.

Thus the tool presented through this paper aims at providing a more general CHR visualization platform. In order to have a flexible tracer, it was decided

M. Proietti and H. Seki (Eds.): LOPSTR 2014, LNCS 8981, pp. 92–110, 2015.
DOI: 10.1007/978-3-319-17822-6_6

to use an already existing visualization tool. Such tools usually provide a wide range of objects and sometimes actions as well. As a proof of concept, we used Jawaa [3] throughout the paper. The annotation tool is available through: http://sourceforge.net/projects/chrvisualizationtool. A web version is also under development and should be available through http://met.guc.edu.eg/chranimation.

In addition to introducing a generic CHR algorithm visualization system, the tool has a module that allows the user to visualize the exhaustive execution of any CHR program forcing the program to produce all the possible solutions. This allows the user to trace the flow of a CHR program using some different semantics than the refined operational semantics [4] embedded in SWI-Prolog. The output of the visualization is a tree showing the different paths of the algorithm's solutions. The tree is the search tree for a specific goal. It is also linked to the visualization tool as shown in Sect. 8.

The paper is organized as follows: Sect. 2 introduces CHR. Section 3 shows some of the related work and why the tool the paper presents is different and needed. Section 4 shows the general architecture of the system. Section 5 introduces the details of the annotation module. The details of the transformation approach are presented in Sect. 6. Section 7 shows an example of the visualization of algorithms implemented through CHR. Section 8 shows how it was possible to transform CHR programs to produce all the possible solutions instead of only one. Finally, we conclude with a summary and directions for future work.

2 Constraint Handling Rules

A CHR program distinguishes between two types of constraints: CHR constraints introduced by the user and built-in constraints [5]. Any CHR program consists of a set of simpagation rules. Each rule has a head, a body and an optional guard. The head of any CHR rule consists of a conjunction of CHR constraints. The guard of a rule is used to set a condition for applying the rule. The guard can thus only contain built-in constraints. The body, on the other hand, can contain both CHR and built-in constraints [5]. A simpagation rule has the form:

$$optional\_rule\_name \ @ \ H_K \setminus H_R \ \Leftrightarrow \ G \mid B.$$

There are two types of head constraints. H_K is the conjunction of CHR constraint(s) that are kept on executing the rule. On the other hand, H_R are the CHR constraint(s) that are removed once the rule is executed. G is the optional guard that has to be satisfied to execute the rule. B is the body of the rule. The constraints in B are added to the constraint store once the rule is executed.

Using simpagation rules, we can distinguish between two types of rules. A *simplification rule* is a simpagation rule with empty H_K. Consequently, the head constraint(s) are removed on executing the rule. It has the following form:

$$optional\_rule\_name \ @ \ H_R \ \Leftrightarrow \ G \mid B.$$

On the other hand, a *propagation rule* is a simpagation rule with empty H_R. Thus, on executing a propagation rule, its body constraints are added to the constraint store without removing any constraint from the store. Its format is:

$$optional\_rule\_name \ @ \ H_K \ \Rightarrow \ G \mid B.$$

The following program extracts the minimum number out of a set of numbers. It consists of one rule: `extract_min @ min(X) \ min(Y) <=> Y>=X | true.`

As seen from the rule, the numbers are inserted through the constraint `min/1`. The rule `extract_min` is executed on two numbers `X` and `Y` if `Y` has a value that is greater than or equal to `X`. `extract_min` removes from the store the constraint `min(Y)` and keeps `min(X)` because it is a simpagation rule. Thus on consecutive executions of the rule, the only number remaining in the constraint store is the minimum one. For example, for the query `min(9), min(7), min(3)`, the rule is applied on `min(9)` and `min(7)` removing `min(9)`. It is then applied on `min(7)` and `min(3)` removing `min(7)` and reaching a fixed point where the rule is no longer applicable. At that point, the only constraint in the store is `min(3)` which is the minimum number.

3 Why "CHRAnimation"?

This section shows the need for the tool and its contribution. As introduced previously, despite of the fact that CHR has developed into a general purpose language, it lacked algorithm visualization and animation tools. Programmers of CHR used SWI-Prolog's "trace" option which produces a textual trace of the executed rules. Attempts focused on visualizing the execution of the rules. The tool provided through [2] is able to visualize the execution of the rules showing which constraints are being added and removed from the store. However, the algorithm the program implements did not affect the visualization in any means. Visual CHR [6] is another tool that is also able to visualize the execution of CHR programs. However, it was directed towards the Java implementation of CHR; JCHR [7]. To use the tool, the compiler of JCHR had to be modified to add the required features. Although [2] could be extended to animate the execution of different algorithms, the need of having static inputs remained due to the inflexibility of the provided tracer. The attempts provided through [8] and [9] also suffered from the problem of being tailored to some specific algorithms.

Thus compared to existing tools for CHR, the strength of the tool the paper presents comes from its ability to adapt to different algorithm classes. It is able to provide a generic algorithm animation platform for CHR programs. The tool eliminates the need to use any driver or compiler directives as opposed to [6,10] since it uses source-to-source transformation. Although the system adopts the concept of *interesting events* used in Balsa [11] and Zeus [12], the new system is much simpler to use. With the previous systems, algorithm animators had to spend a lot of time writing the views and specifying how the animation should take place. With *CHRAnimation*, it is easy for a user to add or change the animation. In addition, the animator could be the developer of the program or any CHR programmer. Thus this eliminates the need of having an animator with whom the developer should develop an animation plan ahead. Consequently, the tool could be easily used by instructors to animate existing algorithms to teach to students. The system provides an interactive tool. In other words, every time a new query is entered, the animation automatically changes. The animations

thus do not have to be prepared in advance to show in a class room for example and are not just movie-based animations that are not influenced by the inputs of users similar to [13].

Unlike the available systems, the user does not need to know about the syntax and details of the visualization system in use. Using source-to-source transformation eliminates this since the programs are automatically modified without the need of manually instructing the code to produce visualizations. The only need is to specify, through the provided user interface, how the constraints should be mapped to visual objects. In Constraint Logic Programming (CLP), the available visualization tools (such as the tools provided through [14] and [15]) focused on the search space and how domains are pruned. Thus to the best of our knowledge, this is the first tool that provides algorithm animation and not algorithm execution visualization for logic programming.

4 System Architecture

The aim of *CHRAnimation* is to have a generic algorithm animation system. The system however should be able to achieve this goal without the need to manually instrument the program to produce the needed visualizations. *CHRAnimation* consists of modules separating the steps needed to produce the animations and keeping the original programs unchanged.

As seen from Fig. 1, the system has two inputs: the original CHR program P in addition to $Annot_{Cons}$, the output of the so-called "Annotation Module".

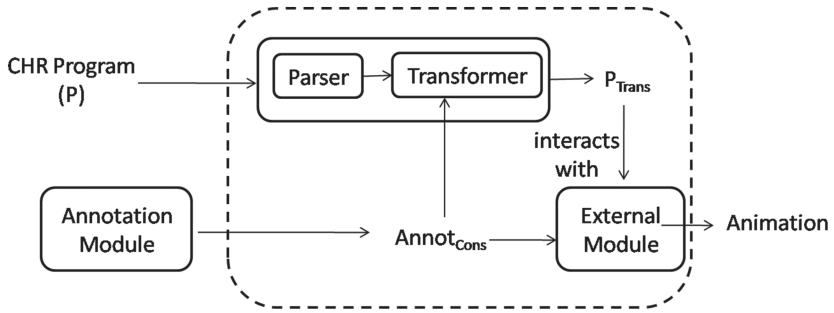


Fig. 1. Interactions between the modules in the system.

As a first step, the CHR program is parsed to extract the needed information. The transformation approach is similar to the one presented in [16] and [2]. Both approaches represent the CHR program using a set of constraints that encode the constituents of the CHR rule. For example `head(extract_min, min(Y), remove)` encodes the information that `min(Y)` is one of the head constraints of the rule named `extract_min` and that this constraint is removed on executing the rule. The CHR program is thus first parsed to automatically extract and

represent the constituents of the rules in the needed "relational normal" form [16]. The *transformer* then uses this representation in addition to $Annot_{Cons}$ to convert the original CHR program (P) to another CHR program (P_{Tran}) with embedded visualization features as explained in more details in Sect. 6.

The *annotation module* is the component that allows the system to animate different algorithms while having a generic visual tracer. It allows users to define the visual states of the algorithm without having to go into any of the actual visualization details. The users are presented with a black-box module which allows them to define the needed visual output through the *interesting events* of the program. The module is explained in more details in Sect. 5. The output of the module ($Annot_{Cons}$) is used by different components of the system to be able to produce the corresponding animation.

P_{Trans} is a normal CHR program that users can run. Whenever the user enters any query to the system, P_{Trans} automatically communicates with an *external module* that uses $Annot_{Cons}$ to spontaneously produce an animation for the algorithm.

5 Annotation to Visualize CHR Algorithms

Algorithm animation represents the different states of the algorithm as pictures. The animation differs according to the interaction between such states [17]. As discussed before, the tool uses an existing tracer to overcome the problems faced in [2] in order to have a dynamic system that could be used with any algorithm type. The *annotation module* is built to achieve this goal while keeping a generic platform that is not tailored according to the algorithm type. Such module is needed to link between the different CHR constraints and the Jawaa objects/-commands. The idea is similar to the "interesting events" that Balsa [11] and Zeus [12] uses. This section introduces the basic functionalities of the annotation module which were first presented in [18] in addition to the new features that were added to accommodate for a wider set of algorithms. In the system, an interesting event is basically defined as the addition of CHR constraint(s) that leads to a change in the state of the algorithm and thus a change in the visualized data structure. For example, in sorting algorithms, every time an element in the list is moved to a new position, the list changes and thus the visualized list should change as well.

5.1 Basic Constraint Annotation

Constraint annotation is the basic building block of the annotation module. Users first identify the interesting events of a program. They could then determine the graphical objects that should be linked to them. For example, the program introduced in Sect. 2 represents a number through the constraint `min/1` with its corresponding value. Adding or changing the `min` constraint is the interesting event in this algorithm. The annotation module provides its users with an interface through which they can choose to link constraint(s) with object(s) and/or action(s) as shown in Fig. 2.

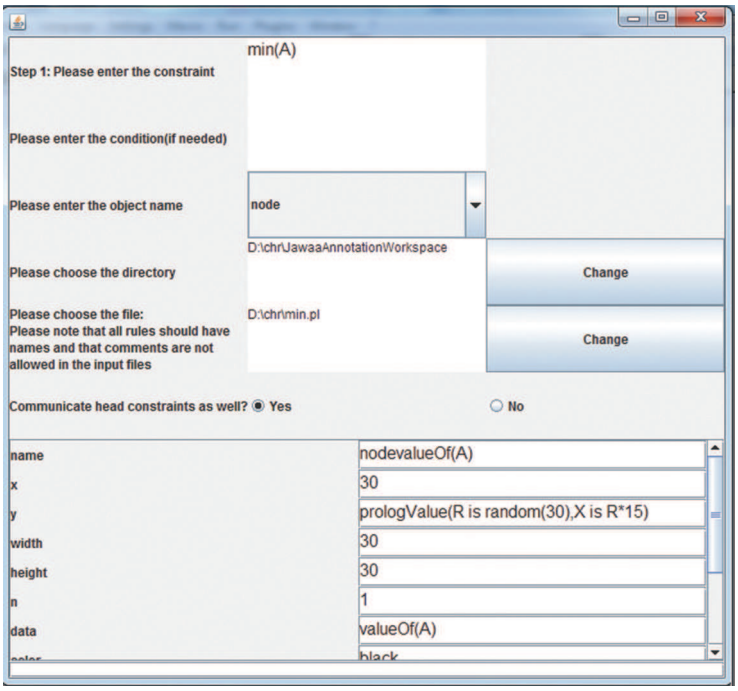


Fig. 2. Annotating the min constraint.

In order to have a dynamic system, the tool is automatically populated through a file that contains the available objects and actions and their corresponding parameters in the form $object\_name\#parameter_1\# \ldots \#parameter_N$. For example, the line `circle#name#x#y#width#color#bkgrd`, adds the object `circle` as one of the available objects to the user. The `circle` object requires the parameters `name`, `x`, `y`, `width`, `color` and `bkgrd`. Users can then enter the name of the constraint and the corresponding annotation as shown in Fig. 2. The current system provides more annotation options than the prototype introduced in [18]. Users enter the constraint: $cons(Arg_1, \ldots, Arg_n)$ representing the interesting event. With the current system, annotations can be activated according to defined conditions. Thus users provide some (Prolog) condition that should hold to trigger the annotation to produce the corresponding visualization. Users can then choose an object/action for annotation. This dynamically fills up the panel with the needed parameters so that users can enter their values. Parameter values can contain one or more of the following values (Val):

1. A constant c. The value can differ according to the type of the parameter. It could be a number, some text, ... etc.
2. The built-in function $valueOf(Arg_i)$ to return the value of an argument (Arg_i).

3. The built-in function $prologValue(Expr)$ where $Expr$ is a *Prolog* expression that binds a variable named X to some value. The function returns the binding value for X.

The output of the constraints' annotations is a list where each element $Cons_{Annot}$ has the form $cons(Arg_1, \ldots, Arg_n) ==> condition\#parameter_1 = Val_1\# \ldots \# parameter_m = Val_m$. In Fig. 2, the user associates the `min/1` constraint with the Jawaa object "Node". In the given example, the name of the Jawaa node is "node" concatenated with the value of the first argument. Thus for the constraint `min(9)`, the corresponding node has the name `node9`. The y-coordinate is random value calculated through Prolog. The text inside the node also uses the value of the argument of the constraint. The annotation is able to produce an animation for this algorithm as shown later in Sect. 7.

5.2 Multi-constraint Annotation

In addition to the basic constraint annotation, users can also link one constraint to multiple visual objects and/or actions. Thus each constraint (*cons*), can add to the output annotations' list multiple elements $Cons_{Annot_1}, \ldots, Cons_{Annot_n}$ if it has n associations.

In addition, users can combine multiple constraints $cons_1, \ldots, cons_N$ in one annotation. This signifies that the interesting event is not associated with having only one constraint in the store. It is rather having all of the constraints $cons_1, \ldots, cons_N$ simultaneously in the constraint store. Such annotations could thus produce and animate a color-mixing program for example. This kind of annotations adds to the annotations' list elements of the form:
$cons_1, \ldots, cons_n ==> annotation\_constraint_{cons_1,\ldots,cons_n}$.

5.3 Rule Annotations

In addition, users can choose to annotate CHR rules instead of only having annotations for constraints. In this case, the interesting event is the execution of the rule as opposed to adding a constraint to the store. This results in adding Jawaa objects and/or actions whenever a specific rule is executed. Thus, whenever a rule is annotated this way, a new step in the visual trace is added on executing the rule. Such annotation adds to the annotations' list an element of the form: $rule_i ==> annotation\_constraint_{rule_i}$. Rule annotations ignore the individual annotations for the constraints since the interesting event is associated with the rule instead. Therefore, it is assumed that the only annotation the user should visualize is the rule annotation since it accounts for all the constraints in the body. An example of this annotation is shown in Sect. 7.

6 Transformation Approach

The transformation mainly aims at interfacing the CHR programs with the entered annotations to produce the needed visual states. Thus the original program P is parsed and transformed into another program P_{Trans}. P_{Trans} performs

the same functionality as P. However, it is able to produce an animation for the executed algorithm for any input query. As a first step, the transformation adds for every constraint `constraint/N` a rule of the form:

$comm\_cons\_constraint$ @ $constraint(X_1, X_2, ..., X_n) \Rightarrow check(status, false) \mid communicate\_constraint(constraint(X_1, X_2, ..., X_n))$.

This extra rule makes sure that every time a new `constraint` is added to the constraint store, it is communicated to the *external module*. Thus, in the case where the user had specified this `constraint` to be an interesting event (i.e. entered an annotation for it), the corresponding object(s)/action(s) is automatically produced. With such new rules, any new constraint added to the store is automatically communicated. Thus once the body constraints are added to the store, they are automatically communicated to the tracer.

The rules of the original program P can affect the visualization through the head constraints. To be more specific, head constraints removed from the store can affect the resulting visualization since their corresponding object(s) might need to be removed as well from the visual trace. Thus the transformer can instruct the new rules to communicate the head constraints[1].

The transformer also makes use of the output of the annotation module output ($Annot_{cons}$). Thus as a second step, the transformer adds for every compound constraint-annotation of the form:
$cons_1, \ldots, cons_n$ ==> $annotation\_constraint_{cons_1,...,cons_n}$, a new rule of the form: $compound_{cons_1,...,cons_n}$ @
$cons_1(Arg_{cons_{1_1}}, \ldots, Arg_{cons_{1_{1x}}}), \ldots, cons_n(Arg_{cons_{n_1}}, \ldots, Arg_{cons_{n_{ny}}})$
$\Rightarrow check(status, false) \mid annotation\_constraint_{cons_1,...,cons_n}(Arg1, \ldots, Arg_m)$.
The default case is to keep the constraints producing a propagation rule but the transformer can be instructed to produce a simplification rule instead. The annotation should be triggered whenever the constraints $cons_1, \ldots, cons_n$ exist in the store producing $annotation\_constraint_{cons_1,...,cons_n}$. This is exactly what the new rules ($compound_{cons_1,...,cons_n}$) do. They add to the store the annotation constraint whenever the store contains $cons_1, \ldots, cons_n$. The annotation constraint is automatically communicated to the tracer through the new $comm\_cons\_constraint\_name$ rules.

As a third step, the rules annotated by the user have to be transformed. The problem with rule annotations is that the CHR constraints in the body should be neglected since the whole rule is being annotated. Thus, even if the constraints were determined by the user to be interesting events, they have to be ignored since the execution of the rule includes them and the rule itself was annotated as an interesting event. Hence, to avoid having problems with this case, a generic *status* is used throughout P_{Trans}. In the transformed program, any rule annotated by the user changes the `status` to *true* at execution. All the new rules added by the transformer to P_{Trans} check that the `status` is set to $false$ before communicating the corresponding constraint to the tracer. Consequently, such

[1] The tracer is able to handle the problem of having multiple Jawaa objects with the same name by removing the old object having the same name before adding the new one. This is possible even if the removed head constraint was not communicated.

rules are not triggered on executing an annotated rule since the guard check is always $false$ in this case. Any rule $rule_i @ H_K \ , \ H_K \ \Leftrightarrow \ G \mid B$ with the corresponding annotation $rule_i ==> annotation\_constraint_{rule_i}$ is transformed to: $rule_i @ H_K \ , \ H_K \ \Leftrightarrow \ G \mid set(status, true), \ B, \ annotation\_constraint_{rule_i}, \ set(status, false)$. In addition, the transformer adds the following rule to P_{Trans}:

$$comm\_cons_{annotation\_constraint_{rule_i}} \ @ \ annotation\_constraint_{rule_i} \ \Leftrightarrow \\ communicate\_constraint(annotation\_constraint_{rule_i}).$$

The new rule thus ensures that the events associated with the rule annotation are considered and that all annotations associated with the constraints in the body of the rule are ignored.

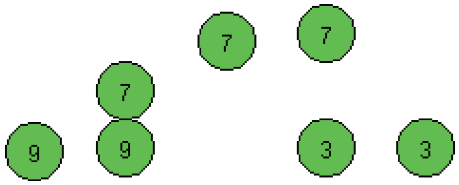
7 Examples

This section shows different examples of how the tool can be used to animate different types of algorithms.

Finding the Minimum Number in a Set is a CHR program consisting of one rule that is able to extract the smallest number out of a set of numbers as shown in Sect. 2. The interesting event in this program is adding and removing the constraint `min`. It was annotated using the basic constraint annotation producing the association: `min(A)==>node##name=nodevalueOf(A)#x=30#y=prologValue(R is random(30), X is R*15)#width=30#height=30#n=1#data=valueOf(A)#color=black#bkgrd=green#textcolor=black#type=CIRCLE`. The annotation links every `min` constraint to a Jawaa "Node" whose y-coordinate is randomly chosen through the function $prologValue$. The x-coordinate is fixed to a constant (30 in our case). As seen in Fig. 3, once a number is added to the store, the corresponding *node* is visualized. Once a number is removed from the store, its *node* object is removed. Thus by applying the rule, `extract_min`, the user gets to see in a step-by-step manner an animation for the program.

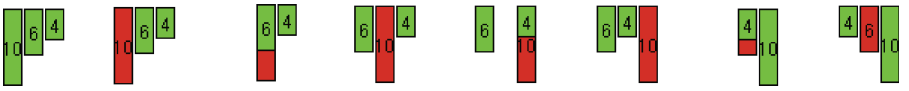
Bubble Sort is another algorithm that could be animated with the tool.

```
start @ totalNum(T)<=> startBubbling, loop(1,1,T).
```



(a) adding min(9) to the store
(b) adding min(7) to the store
(c) removing min(9)
(d) adding min(3) to the store
(e) removing min(7)

Fig. 3. Finding the minimum element of a set.



(a) all elements inserted to the list (b) highlighting 10 (c) moving 10 to the right (d) moving 6 to the left (e) moving 10 to the right (f) moving 4 to the left (g) highlighting 6 and moving it to the right (h) moving 4 to the left

Fig. 4. Sorting a list of numbers using rule annotations.

(a) all elements inserted to the list (b) 10, 6 removed (c) 10, 6 swapped (d) 10, 4 removed (e) 10, 4 swapped (f) 6, 4 removed (g) 6, 4 swapped

Fig. 5. Sorting a list of numbers using constraint annotations only.

```
bubble @ startBubbling, loop(I,_,_) \ a(I,V), a(J,W) <=> I+1=:=J, V>W |
                                                        a(I,W), a(J,V).
loop1 @ startBubbling\ loop(A,B,C) <=> A<C, B<C | A1 is A+1, loop(A1,B,C).
loop2 @ startBubbling \ loop(C,B,C) <=> B<C | B1 is B+1, loop(1,B1,C).
```

As seen from the program, the different elements of the list are entered using the constraint `a/2`. The rule `bubble` swaps two consecutive elements that are not sorted with respect to each other. Consequently, through multiple executions of this rule, the largest element is bubbled to the end. The constraint `loop/3` represents a pointer to the elements being compared. `loop1` advances the pointer one step through the list. `loop2` resets the pointer to the beginning of the list whenever one complete round of checks is done. The bubbling step is repeated T times where T is the number of elements in the list. There are thus two interesting events in this program. The first one is the insertion of an element to the list which is represented by the constraint `a/2`. The second interesting event is swapping two consecutive elements together through the rule `bubble`. The program has *three* annotations. The first one is a basic constraint annotation for `a/2` constraint. The second annotation is a rule annotation for `bubble`. The

rule is annotated with `swap/4` which has as arguments `I,V,J` and `W` consecutively. `swap/4` has a multi-constraint annotation that does the following:

1. highlights the element at index `I` through a "changeParam" action,
2. moves the element at index `I` to the right through a `moveRelative` action,
3. moves the element at index `J` to the left through a `moveRelative` action.

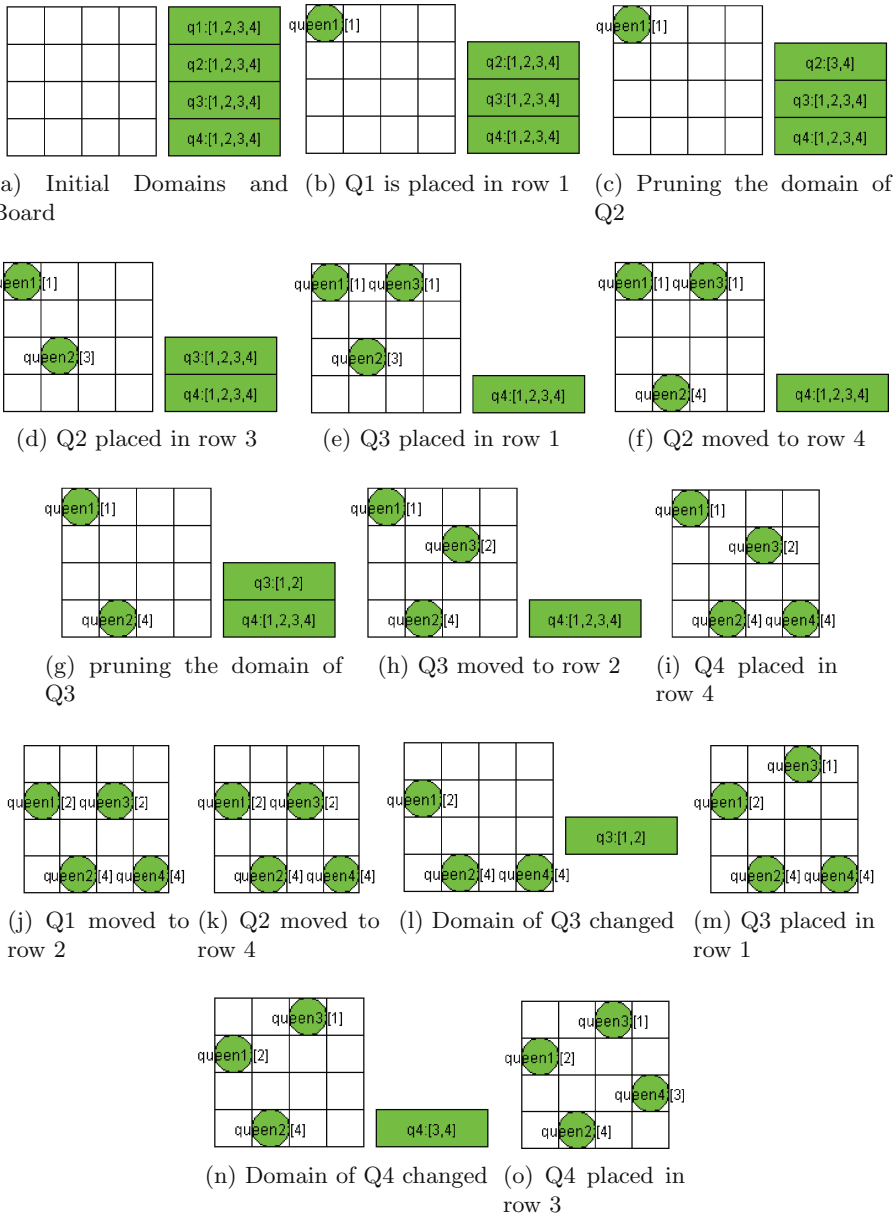
The annotations are shown in Appendix A. The output animation for the query (`a(1,10),a(2,6),a(3,4),totalNum(3)`) is given in Fig. 4. Figure 5 shows the result if no rule annotation was used. In this case, the different nodes representing the elements in the list are added and removed.

N**queens** is a well-known problem in which N queens have to be placed on an N by N grid such that they do not attack each other. Two queens can attack each other if they are placed on the same row, column or diagonal. The following CHR program can solve this problem:

```
initial @ solve(N) <=> generate(1,N,List), queens(N,List), labelq.
add1 @ queens(N,Dom) <=> N>0 | N1 is N-1, in(N , Dom), queens(N1,Dom).
add2 @ queens(0,Dom) <=> true.
reduce @ in(N1 , [P]) \ in(N2 , Dom) <=> P1 is P-(N1-N2),
               P2 is P+(N1-N2), delete(Dom,P,D1), delete(D1,P1,D2),
                  delete(D2,P2,D3),Dom\==D3 | D3\==[], in(N2 , D3).
label @ labelq \ in(N , Dom)  <=> Dom=[\_,\_|\_] | member(P,Dom), in(N , [P]).
```

The model of the problem uses `N` variables each represented using the `queens/2` constraint. The value of every variable determines the row number. The index, on the other hand, determines the column number. For example if the value of the second queen is three, this means that the queen in the second column is placed in the third row. The domain of any queen is initialized to be from 1 to `N` using the predicate `generate/3`. As seen from the program the rule `initial` is used to initialize the solving process by adding the two constraints `queens/2` and `labelq` which enable finding a solution. As seen from the rule, the second argument of the `queens` constraint is set to be a list containing all the numbers from `1` till `N`. The two rules `add1` and `add2` are used to initialize the domains of all of the queens of the board using the previously computed list. The domain of every queen is represented using the `in/2` constraint. The rule `reduce` is used to prune the domains of the different queens. In order to execute the rule, the location of a specific queen has to be determined. This is represented by having a domain list with one element only. The rule removes from the domain of another queen any value that could lead to an attack. This ensures that whenever a location is chosen for this queen, it does not threaten the already labeled queen. Finally the rule `label` is used to search through the domains whenever domain pruning is not enough.

The visual board is initialized through specifying that the `solve` constraint is an interesting event. It should generate 16 rectangles in a board-like structure. To eliminate the need of entering 16 constraint annotations, users can now use the object *board* to annotate constraints and enter the number of squares it contains and their widths, heights, ... etc. Thus whenever the `solve` constraint is added



(a) Initial Domains and Board (b) Q1 is placed in row 1 (c) Pruning the domain of Q2

(d) Q2 placed in row 3 (e) Q3 placed in row 1 (f) Q2 moved to row 4

(g) pruning the domain of Q3 (h) Q3 moved to row 2 (i) Q4 placed in row 4

(j) Q1 moved to row 2 (k) Q2 moved to row 4 (l) Domain of Q3 changed (m) Q3 placed in row 1

(n) Domain of Q4 changed (o) Q4 placed in row 3

Fig. 6. Visualizing the execution of the *n*queens algorithm for 4 queens.

to the store the 4-by-4 grid is visualized. The board, in the 4-queens problem, consists of `16` adjacent rectangles each with the same width and height (30 was chosen in this example). The `in/2` constraint has two different annotations. The first one is activated whenever the length of the domain list is equal to *one*. This is the case where the queen is labeled to be placed in a specific position on the board. In this case, the x-coordinate of the Jawaa node is calculated as the index multiplied by the width of the cell which is `30`. The y-coordinate is calculated through the only value in the domain i.e. the assigned value. It is also multiplied by `30`. This way the circular node is placed in the a location corresponding to the chosen value. The second annotation is activated whenever the length of the domain list is greater than *one*. In this case the queen is not placed in any position in the board since there are multiple possibilities. It is visualized as a "Node" outside the board and the domain is written on it. Figure 6 shows the visual steps produced for the query `solve(4)` until a solution is found. The annotations are shown in Appendix A.

8 Visualizing Different Semantics

Although SWI-Prolog implements the refined operational semantics [4] for CHR, there are different proposed and defined CHR operational semantics. Based on the conflict resolution approach presented in [5], it is possible to convert a program running with a different operational semantics into the refined operational semantics used in SWI-Prolog. The abstract operational semantics of CHR [5] is nondeterministic. At any point, if several rules are applicable, one of them is randomly chosen. The application of a rule, however, cannot be undone since it is committed choice. In addition, the goal constituents are randomly chosen for processing. The refined operational semantics [4], on the other hand, chooses a top-bottom approach for deciding on the applicable rule i.e. the first applicable rule is always chosen. In addition, the constraints are processed from left to right. Nondeterminism is especially interesting when the CHR program is non-confluent. Confluence [19] is a property of CHR which ensures the same final result no matter which applicable rule was chosen at any point of the execution. The tool includes a module that is able to embed some of the nondeterminism properties into any CHR solver. The newly generated solvers are able to choose, at any point of the execution, any of the applicable rules producing all the possible solutions.

8.1 Transformation Approach

This section discusses how any CHR program is transformed into a new one that is able to generate all the possible solutions instead of using the refined operational semantics that generates only one solution. The transformation approach is based on the approaches presented in [5,20–22]. The main difference is that the new solver communicates some of the information to the visual tracer to be able to produce the needed visualization.

The transformed program starts each step by collecting the set of applicable rules with its corresponding head constraints. After the candidate list is built, the solver chooses one of the rules randomly using the built-in predicate `select/3`. The newly transformed program is thus a CHR˜ [23] solver. For example a rule of the form:

```
r1 @ Hk \ Hr <=> Guard | Body.
```

generates two rules in the transformed program. The first generated rule is used to populate the candidate list. It is a propagation rule of the form:

```
Hk, Hr ==> Guard | cand([(r1,[Hk,Hr])]).
```

The second rule is fired whenever this rule is chosen from the candidate list. It has the following form:

```
Hk\fire((r1,[Hk,Hr])),Hr <=> Guard | communicate_heads_kept(Hk),
        communicate_heads_removed(Hr),communicate_body(Body),Body.
```

In addition, the new program contains the following two rules:

```
cand(L1),cand(L2) <=> append(L1,L2,L3) | cand(L3).
cand([H|T]),fire <=> select(Mem,[H|T],Nlist), fire(Mem),cand(NList),fire.
```

The first rule ensures that the candidate list is correctly populated and incremented. The second rule, on the other hand, selects one of the elements of the candidate list at each step.

For example the program:

```
:-chr_constraint sphere/2.
r1 @ sphere(X,red) <=> sphere(X,blue).
r2 @ sphere(X,red) <=> sphere(X,green).
```

is transformed into

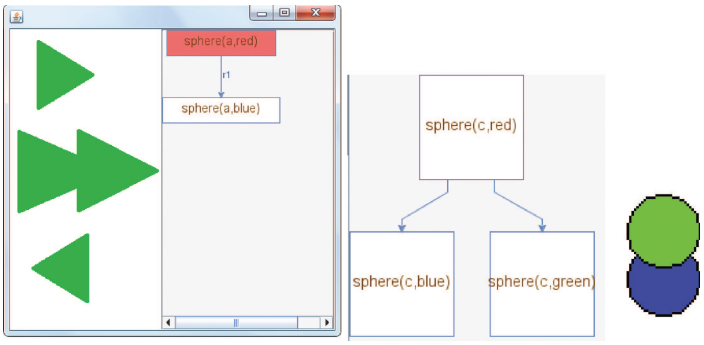
```
:-chr_constraint sphere/2, fire/1, cand/1, fire/0.
r1_cand @ sphere(X,red) ==>cand([(r1,[sphere(X,red)])]).
r2_cand @ sphere(X,red) ==> cand([(r2,[sphere(X,red)])]).
cand(L1),cand(L2) <=> append(L1,L2,L3), cand(L3).
cand([H|T]),fire <=>  select(Mem,[H|T] , NList), fire(Mem), cand(NList),fire.
r1 @ fire((r1,[sphere(X,red)])),sphere(X,red) <=>
                  communicate_head_removed([sphere(X,red)]),
         communicate_body([sphere(X,blue)]), sphere(X,blue).
r2 @ fire((r2,[sphere(X,red)])),sphere(X,red) <=>
                  communicate_head_removed([sphere(X,red)]),
        communicate_body([sphere(X,green)]), sphere(X,green).
```

8.2 Visualization

With the refined operational semantics, the query `sphere(a,red)` results in executing `r1` adding to the store the new constraint `sphere(a,blue)`. Figure 7a shows the result of visualizing the execution of the solver with this query, using the tool presented in [2]. The CHR constraints remaining in the constraint store are shown in white and those removed are shown in red. The transformed program is able to generate the visual tree shown in Fig. 7b. Since there were two applicable rules, the output tree accounts for both cases by the different paths. Through SWI-Prolog the user can trigger this behavior using the ";" sign to search for more solutions.

Given the solver:
`rule1 @ sphere(X,red) <=> sphere(X,blue).`
`rule2 @ sphere(X,blue) <=> sphere(X,green).` The steps taken to execute the query `sphere(b,blue), sphere(a,red)` with the solver are:

- First Solution
  1. `rule2` is fired replacing the constraint `sphere(b,blue)` by `sphere(b,green)`.
  2. `rule1` is then fired removing the constraint `sphere(a,red)` and adding the constraint `sphere(a,blue)`.
  3. Finally, `sphere(a,blue)` triggers `rule2` replacing it by `sphere(a,green)`.
- Second Solution
  1. Backtracking is triggered through the semicolon(`;`). We thus go back to the root and choose to apply `rule1` for `sphere(a,red)` producing the `sphere(a,blue)`.
  2. Afterwards, `rule2` is executed to replace `sphere(a,blue)` by `sphere(a,green)`.
  3. Finally, `rule2` is fired replacing `sphere(b,blue)` by `sphere(b,green)`.
- Third Solution
  1. This time when the user backtracks, execution goes back to the second level, applying `rule2` to replace `sphere(b,blue)` by `sphere(b,green)`.
  2. Afterwards, `rule2` replaces `sphere(a,blue)` by `sphere(a,green)`.



(a) Visualizing the execution of the solver. (b) Showing all possible paths. (c) Two random circles.

Fig. 7. Different options for visualizing the execution.

As seen from the tree in Fig. 8, the constraint store in the final states contains `sphere(a,green), sphere(b,green)`. However, the paths taken are different. Once the user enters a query, the visual trees are automatically shown. In addition, whenever the user clicks on any node in the tree, the corresponding visual annotations are triggered. In this case the sphere can be mapped to a

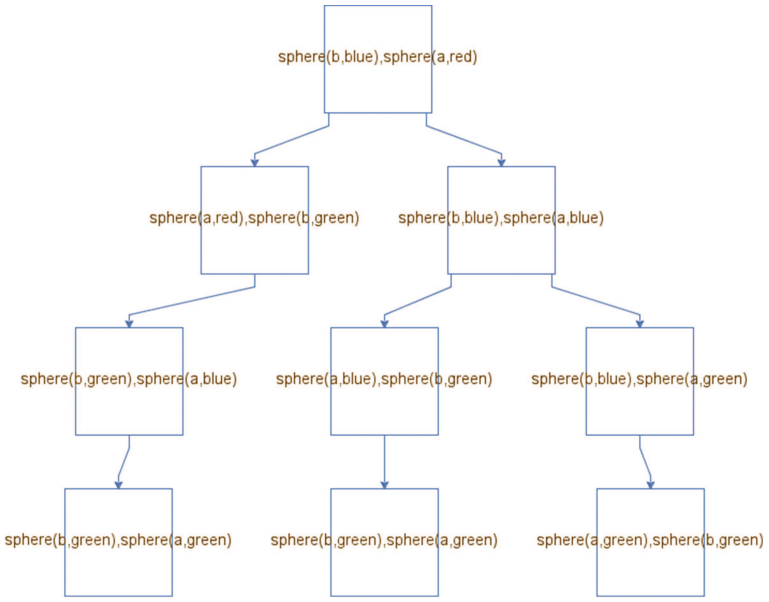


Fig. 8. Output tree.

Jawaa "circle" with a constant x-coordinate and a random y-coordinate and a background color that is equal to the value of the second argument of the constraint. If the user clicks on the node with the constraints (`sphere(b,blue)`, `sphere(a,green)`), the system automatically connects the constraints to the previously introduced visual tracer that checks if any of the current constraints have annotations. This produces a visual state with two circles placed randomly as shown in Fig. 7c.

9 Conclusion

The paper introduced a new tool that is able to visualize different CHR programs by dynamically linking CHR constraints to visual objects. To have a generic tracing technique, the new system outsources the visualization process to existing tools. Intelligence is shifted to the transformation and annotation modules. Through the provided set of visual objects and actions, different algorithms could be animated. Such visualization features have proven to be useful in many situations including code debugging for programmers and educational purposes [24]. In addition, the paper explores the possibility of visualizing the execution of different operational semantics of CHR. It provides a module that is able to visualize the exhaustive execution of CHR and more importantly it links it to the annotated constraints. Thus, unlike the previously provided tools [15] for visualizing constraint programs, the focus is not just on the search space and the

domains. The provided tool enables its users to focus on the algorithms executed to visualize their states.

In the future, more dynamic annotation options could be provided to the user. The visualization of the execution of different CHR operational semantics should be investigated. The tool could also be extended to be a visual confluence checker for CHR programs. In addition, we also plan to investigate the visualization of soft constraints [25].

Appendix

A Annotations

The bubble sort program's annotations use *node* as a basic object. The x-coordinate is calculated through the index and the height uses the value.

```
a(Index,Value)==>node##name=nodevalueOf(Index)#x=valueOf(Index)*14+2#y=100#
width=12#height=valueOf(Value)*5#n=1#data=valueOf(Value)#color=black#
bkgrd=green#textcolor=black#type=RECT
swap(I1,V1,I2,V2)==>changeParam##name=nodevalueOf(I1)#paramter=bkgrd #newvalue=red
swap(I1,V1,I2,V2)==>moveRelative##name=nodevalueOf(I1)#x=14#y=0
swap(I1,V1,I2,V2)==>moveRelative##name=nodevalueOf(I2)#x=-14#y=0
swap(I1,V1,I2,V2)==>changeParam##name=nodevalueOf(I1)#paramter=bkgrd #newvalue=green
```

For the $n$queens problem, the first set of annotations are the rectangles produced by the object "board" which users can choose through the interface. The number of vertical and horizontal squares (4 in our case), the initial x and y-coordinates (30 and 30 in our case), the squares' widths (30 in this case) in addition to the color chosen by the user automatically produces 16 associations for the `solve` constraint. The below constraints represent the rectangles that form the first two rows of the board and the two annotations for the `in` constraint.

```
solve(N)==>rectangle##name=rect1#x=30#y=30#width=30#height=30#color=black#bkgrd=white
solve(N)==>rectangle##name=rect2#x=60#y=30#width=30#height=30#color=black#bkgrd=white
solve(N)==>rectangle##name=rect3#x=90#y=30#width=30#height=30#color=black#bkgrd=white
solve(N)==>rectangle##name=rect4#x=120#y=30#width=30#height=30#color=black#bkgrd=white
in(N,List)==>node#length(valueOf(List),Len),Len is 1#name=nodevalueOf(N)#x=valueOf(N)*30
#y=prologValue(nth0(0,valueOf(List),El),X is El*30)#width=30#height=30#n=1#
data=queenvalueOf(N):valueOf(List)#color=black#bkgrd=green#textcolor=black#type=CIRCLE
in(N,List)==>node#length(valueOf(List),Len),Len > 1#name=nodevalueOf(arg0)#x=160#
y=valueOf(N)*30#width=90#height=30#n=1#data=qvalueOf(N):valueOf(List)#color=black#
bkgrd=green#textcolor=black#type=RECT
```

References

1. Frühwirth, T.: Theory and practice of constraint handling rules, special issue on constraint logic programming. J. Logic Program. **37**, 95–138 (1998)
2. Abdennadher, S., Sharaf, N.: Visualization of CHR through source-to-source transformation. In: Dovier, A., Costa, V.S. (eds.) ICLP (Technical Communications). LIPIcs, vol. 17, pp. 109–118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)

3. Rodger, S.H.: Introducing computer science through animation and virtual worlds. In: Gersting, J.L., Walker, H.M., Grissom, S. (eds.) SIGCSE, pp. 186–190. ACM (2002)
4. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
5. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (2009)
6. Abdennadher, S., Saft, M.: A visualization tool for constraint handling rules. In: Kusalik, A.J. (ed.) WLPE (2001)
7. Schmauss, M.: An Implementation of CHR in Java, Master Thesis, Institute of Computer Science, LMU, Munich, Germany (1999)
8. Ismail, A.: Visualization of Grid-based and Fundamental CHR Algorithms, bachelor thesis, the Institute of Software Engineering and Compiler Construction, Ulm University, Germany (2012)
9. Said, M.A.: Animation of Mathematical and Graph-based Algorithms expressed in CHR, bachelor thesis, the Institute of Software Engineering and Compiler Construction, Ulm University, Germany (2012)
10. Stasko, J.: Animating algorithms with xtango. SIGACT News **23**, 67–71 (1992)
11. Brown, M.H., Sedgewick, R.: A system for algorithm animation. In: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, pp. 177–186. ACM, New York (1984)
12. Brown, M.: Zeus: a system for algorithm animation and multi-view editing. In: Proceedings of the 1991 IEEE Workshop on Visual Languages, pp. 4–9 (1991)
13. Baecker, R.M.: Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science, chap. 24, pp. 369–381. MIT Press, Cambridge (1998)
14. Smolka, G.: The definition of kernel oz. In: Podelski, A. (ed.) Constraint Programming: Basics and Trends. LNCS, vol. 910. Springer, Heidelberg (1995)
15. Meier, M.: Debugging constraint programs. In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976. Springer, Heidelberg (1995)
16. Frühwirth, T., Holzbaur, C.: Source-to-source transformation for a class of expressive rules. In: Buccafurri, F. (ed.) APPIA-GULP-PRODE, pp. 386–397 (2003)
17. Kerren, A., Stasko, J.T.: Algorithm animation. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 1–17. Springer, Heidelberg (2002)
18. Sharaf, N., Abdennadher, S., Frühwirth, T. W.: Visualization of Constraint Handling Rules, CoRR, vol. abs/1405.3793 (2014)
19. Abdennadher, S., Frühwirth, T., Meuss, H.: On confluence of constraint handling rules. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118. Springer, Heidelberg (1996)
20. Zaki, A., Frühwirth, T.W., Abdennadher, S.: Towards inverse execution of constraint handling rules. TPLP **13**(4-5) (2013) Online-Supplement
21. Abdennadher, S., Fakhry, G., Sharaf, N.: Implementation of the operational semantics for CHR with user-defined rule priorities. In: Christiansen, H., Sneyers, J. (eds.) Proceedings of the 10th Workshop on Constraint Handling Rules, pp. 1–12, Technical report CW 641, (2013)
22. Fakhry, G., Sharaf, N., Abdennadher, S.: Towards the implementation of a source-to-source transformation tool for CHR operational semantics. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013, LNCS 8901. LNCS, vol. 8901, pp. 145–163. Springer, Heidelberg (2014)

23. Abdennadher, S., Schütz, H.: CHRv: a flexible query language. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS (LNAI), vol. 1495, pp. 1–14. Springer, Heidelberg (1998)
24. Hundhausen, C., Douglas, S., Stasko, J.: A meta-study of algorithm visualization effectiveness. J. Vis. Lang. Comput. **13**(3), 259–290 (2002)
25. Bistarelli, S., Frühwirth, T., Marte, M.: Soft constraint propagation and solving in chrs. In: Proceedings of the 2002 ACM Symposium on Applied Computing, pp. 1–5. ACM (2002)