CHR^{vis}: Syntax and Semantics

Nada Sharaf, Slim Abdennadher
The German University in Cairo
{nada.hamed,slim.abdennadher}@guc.edu.eg

Thom Frühwirth
Ulm University
thom.fruehwirth@uni-ulm.de

Abstract

The work in the paper presents an animation extension (CHR^{vis}) to Constraint Handling Rules (CHR). Visualizations have always helped programmers understand data and debug programs. A picture is worth a thousand words. It can help identify where a problem is or show how something works. It can even illustrate a relation that was not clear otherwise. CHR^{vis} aims at embedding animation and visualization features into CHR programs. It thus enables users, while executing programs, to have such executions animated. The paper aims at providing the operational semantics for CHR^{vis}. The correctness of CHR^{vis} programs is also discussed.

2012 ACM Subject Classification Visualization systems and tools

Keywords and phrases Constraint Handling Rules Visualization Animation

Digital Object Identifier 10.4230/OASIcs.ICLP.2018.5

1 Introduction

Animation tools are considered as a basic construct of programming languages. They are used to visualize the execution of a program. They provide users with a simple and intuitive method to debug and trace programs. This paper presents an extension to Constraint Handling Rules (CHR). The extension adds new visual features to CHR. It enables users to animate executions of CHR programs.

CHR [8, 7] has evolved over the years into a general purpose language. Originally, it was proposed for writing constraint solvers. Due to its declarativity, it has, however, been used with different algorithms such as sorting algorithms, graph algorithms, ... etc. CHR lacked tracing and debugging tools. Users were only able to use the textual trace facility of SWI-Prolog as shown in Figure 1 which is hard to follow especially with big programs.

Two types of visual facilities are important for a CHR programmer/beginner. Firstly, the programmer would like to get a visual trace showing which CHR rule gets applied at every step and its effect.

```
[trace] 1 ?- min(4),min(1),min(10).
   Call: (7) min(4) ? creep
^  Call: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
235), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
CHR:   (0) Insert: min(4) # <71>
^  Exit: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
235), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
   Exit: (7) min(4) ? creep
   Call: (7) min(1) ? creep
^  Call: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
430), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
CHR:   (0) Insert: min(1) # <72>
^  Exit: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
430), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
```

(a) Using the normal trace option

```
2 ?- min(4),min(1),min(10).
CHR:   (0) Insert: min(4) # <71>
CHR:   (1) Call: min(4) # <71> ? [creep]
CHR:   (1) Exit: min(4) # <71> ? [creep]
CHR:   (0) Insert: min(1) # <72>
CHR:   (1) Call: min(1) # <72> ? [creep]
CHR:   (1) Try: min(1) # <72> \ min(4) # <71> <=> 1<_G31917 | true.
CHR:   (1) Apply: min(1) # <72> \ min(4) # <71> <=> 1<_G31917 | true. ? [creep]
CHR:   (1) Remove: min(4) # <71>
CHR:   (1) Exit: min(1) # <72> ? [creep]
CHR:   (0) Insert: min(10) # <73>
CHR:   (1) Call: min(10) # <73> ? [creep]
CHR:   (1) Try: min(1) # <72> \ min(10) # <73> <=> 1<10 | true.
CHR:   (1) Apply: min(1) # <72> \ min(10) # <73> <=> 1<10 | true. ? [creep]
CHR:   (1) Remove: min(10) # <73>
CHR:   (1) Exit: min(10) # <73> ?
```

(b) Using the chr_trace option

Figure 1 Current Tracing Facilities in SWI-Prolog.

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 5; pp. 5:1–5:19
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Secondly, since CHR has developed into a general purpose language, it has been used with different types of algorithms such as sorting and graph algorithms. It is thus important to have a visual facility to animate the execution of the algorithms rather than just seeing the rules being executed. CHR lacked such a tool. The tool should be able to adapt with the execution nature of CHR programs where constraints are added and removed continuously from the constraint store.

Several approaches have been devised for visualizing CHR programs and their executions. In [1], a tool called *VisualCHR* was proposed. VisualCHR allows its users to visually debug constraint solving. The compiler of JCHR [12] (on which VisualCHR is based) was modified. The visualization feature was thus not available for Prolog versions, the more prominent implementation of CHR. [2] introduced a tool for visualizing the execution of CHR programs. It was able to show at every step the constraint store and the effect of applying each CHR rule in a step-by-step manner. The tool was based on the SWI-Prolog implementation of CHR. Source-to-source transformation was used in order to eliminate the need of doing any changes to the compiler. The tool could thus be deployed directly by any user.

Despite of the availability of such visualization tools, CHR was still missing a system for animating algorithms. The available tools were able to show at each point in time the executed rule and the status of the constraint store [2, **?**]. However, the algorithm implemented had no effect on the produced visualization. Existing algorithm animation tools could not be adopted with CHR. For example, one of the available tools is XTANGO [15] which is a general purpose animating system. However, the algorithm should be implemented in C or another language such that it produces a trace file to be read by a C program driver making it difficult to use with CHR. Due to the wide range of algorithms implemented through CHR, an algorithm-based animation was needed. Such animation should show at each step in time the changes to the data structure affected by the algorithm.

The paper presents a different direction for animating CHR programs. It allows users to animate any kind of algorithm implemented in CHR. This direction thus augments CHR with an animation extension. As a result, it allows a CHR programmer to trace the program from an algorithmic point of view independent of the details of the execution of its rules. The formal analysis of the new extension is presented in the paper. The paper thus presents a new operational semantics of CHR that embeds visualization into its execution. The formalism is able to capture not only the behavior of the CHR rules, it is also able to represent the graphical objects associated with the animation. It is used to prove the correctness of the programs extended with animation features. To eliminate the need of users learning the new syntax for using the extension, a transformation approach is also provided.

The paper is organized as follows: Section 2 introduces CHR. Section 3 introduces the new extension. Finally, in Section 3.2 the formalization is given by introducing ω_{vis}, a new operational semantics for CHR that accounts for annotation rules. Conclusions and directions for future work are presented at the end of the paper.

2 Constraint Handling Rules

CHR was initially developed for writing constraint solvers [8, 7, 9]. The rules of a CHR program keeps on rewriting the constraints in the constraint store until a fixed point is reached. At that point no CHR rules could be applied. The constraint store is initialized by the constraints in the query of ths user. CHR has implementations in different languages such as Java, C and Haskell. The most prominent implementation is the Prolog one. A CHR program has two types of constraints: user-defined/CHR constraints and built-in constraint. CHR constraints are defined by the user at the beginning of a program. Built-in constraints, on the other hand, are handled by the constraint theory ($\mathcal{CT}$) of the host language. A CHR program consists of a set of “simpagation rules". A simpagation rule has the following format:

$$optional\_rule\_name \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B.$$

H_k and H_r represent the head of the rule. The body of the rule is B. The guard G represents a precondition for applying the rule. A rule is only applied if the constraint store contains constraints that match the head of the rule and if the guard is satisfied. As seen from the previous rule, the head has two parts: H_k and H_r. The head of a rule could only contain CHR constraints. The guard should consist of built-in constraints. The body, on the other hand, can contain CHR and built-in constraints. On applying the rule, the constraints in H_k are kept in the constraint store. The constraints in H_r are removed from the constraint store. The body constraints are added to the constraint store.

There are two special kinds of CHR rules: propagation rules and simplification rules. A propagation rule has an empty H_r. A propagation rule does not remove any constraint from the constraint store. It has the following format:

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

A simplification rule on the other hand has an empty H_k. A simplification rule removes all the head constraints from the constraint store. A simplification rule has the following format:

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

The following program aims at sorting numbers in an array/list. Each number is represented by the constraint `cell(I,V)`. `I` represents the index and `V` represents the value of the element. The program contains one rule: `sort_rule`. It is applied whenever the constraint store contains two `cell` constraints representing two unsorted elements. The guard makes sure that the two elements are not sorted with respect to each other. The element at index `I1` has a value (`V1`) that is greater than the value (`V2`) of the element at index `I2`. `I1` is less than `I2`. Thus, `V1` precedes `V2` in the array despite of the fact that it is greater than it. Since `sort_rule` is a simplification rule, the two constraints representing the unsorted elements are removed from the constraint store. Two `cell` constraints are added through the body of the rule to represent the performed swap to sort the two elements. Successive applications of the rule makes sure that any two elements that are not sorted with respect to each other are swapped. The fixed point is reached whenever `sort_rule` is no longer applicable. At this point, the array is sorted. The program is shown below:

```
:-chr_constraint cell/2.
sort_rule @ cell(In1,V1), cell(In2,V2) <=> In1<In2,V1>V2 |
                                        cell(In2,V1), cell(In1,V2).
```

2.1 Refined Operational Semantics ω_r

In the theoretical semantics of CHR (ω_t), a state is represented by the tuple $\langle G,S,B,T\rangle_n^V$ [8, 3]. G represents the goal store. It initially contains the query of the user. S is the CHR constraint store containing the currently available CHR constraints. B, on the other hand, is the built-in store with the built-ins handled by the host language (Prolog in this case). The propagation history, T, holds the names of the applied CHR rules along with the identifiers of the CHR constraints that activated the rules. T is used to eliminate the trivial nontermination problem. Each CHR constraint is associated with an identifier. n represents the next available identifier. V represents the set of global variables. Such variables are the ones that exist in the initial query of the user. V does not change during execution, it is thus omitted throughout the rest of the paper. A variable $v \notin V$ is called a local variable [11].

▶ Definition 1. The function `chr` is defined such that `chr(c#n) = c`. It is extended into sequences and sets of CHR constraints. Likewise, the function `id` is defined such that `id(c#n) = n`. It is also extended into sequences and sets of CHR constraints.

The refined operational semantics [6, 8] is adapted in most implementations of CHR. It removes some of the sources of the non-determinism that exists in the theoretical operational semantics (w_t). In w_t the order in which constraints are processed and the order of rule application is non-deterministic. However, in w_r, rules are executed in a top-down manner. Thus, in the case where there are two matching rules, w_r ensures that the rule that appears on top is executed. Each atomic head constraint is associated with a number (occurrence). Numbering starts from 1. It follows a top-down approach as well. For example, the previously shown program to find the minimum value is numbered as follows:

```
remove_dup @ min(X)_2 \ min(X)_1 <=> true.
remove_min @ min(X)_4 \ min(Y)_3 <=> X<Y | true.
```

▶ Definition 2. The active/occurrenced constraint $c\#i : j$ refers to a numbered constraint that should only match with occurrence j of the constraint c inside the program. i is the identifier of the constraint [6].

A state in w_r is the tuple $< A,S,B,T >_n$. Unlike w_t, the goal A is a stack instead of a multi-set. S,B,T and n have the same interpretation as an w_t state. In the refined operational semantics, constraints are executed similar to procedure calls. Each constraint added to the store is activated. An active constraint searches for an applicable rule. The rule search is done in a top-down approach. If a rule matches, the newly added constraints (from the body of the applied rule) could in turn fire new rules. Once all rules are fired, execution resumes from the same point. Constraints in the constraint store are reconsidered/woken if a newly added built-in constraint could affect them (according to the wakeup policy). An active constraint thus tries to match with all the rules in the program. Table 1 shows the transitions of w_r. The explanation of the transitions is given in the Appendix.

1. **Solve+wakeup** : $\langle [c\|A], S_0 \cup S_1, B, T\rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, B', T\rangle_n$ given that c is a built-in constraint and $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$ and $wakeup(S_0 \cup S_1, c, B) = S_1$
2. **Activate** $\langle [c\|A], S, B, T\rangle_n \mapsto_{activate} \langle [c\#n : 1\|A], c\#n \cup S, B, T\rangle_{n+1}$ given that c is a CHR constraint.
3. **Reactivate** $\langle [c\#i\|A], S, B, T\rangle_n \mapsto_{reactivate} \langle [c\#i : 1\|A], S, B, T\rangle_n$ given that c is a CHR constraint.
4. **Apply** $\langle [c\#i : j\|A], H_1 \cup H_2 \cup S, B, T\rangle_n \mapsto_{apply\ r}$ $\langle C + H + A, H_1 \cup S, chr(H_1) = (H_1') \wedge chr(H_2) = (H_2') \wedge g \wedge B, T \cup \{(r, id(H_1) + id(H_2))\}\rangle_n$ given that the jth occurrence of c is part of the head of the re-named apart rule with variables x': r @ $H_1' \setminus H_2' \Leftrightarrow g \mid C$. where $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x' \left((chr(H_1) = (H_1') \wedge chr(H_2) = (H_2') \wedge g)\right)$ and $(r, id(H_1) + id(H_2)) \notin T$. If c occurs in H_1' then $H = [c\#i : j]$ otherwise $H = []$.
5. **Drop** $\langle [c\#i : j\|A], S, B, T\rangle_n \mapsto_{drop} \langle A, S, B, T\rangle_n$ given that $c\#i : j$ is an occurrenced active constraint and c has no occurrence j in the program. That could thus imply that all existing occurrences were tried before.
6. **Default** $\langle [c\#i : j\|A], S, B, T\rangle_n \mapsto_{default} \langle [c\#i : j+1\|A], S, B, T\rangle_n$ in case there is no other applicable transition.

Table 1 Transitions of ω_r

- Solve+Wake: This transition introduces a built-in constraint c to the built-in store. In addition, all constraints that could be affected by c ($S1$) are woken up by adding them on top of the stack. These constraints are thus re-activated. A constraint where all its terms have become ground will not be thus woken up by the implemented wake-up policy since it is never affected by a new built-in constraint. $vars(S_0) \subseteq fixed(B)$ where $fixed(B)$ represents the variables fixed by B.
- Activate: This transition introduces a CHR constraint into the constraint store and activates it. The introduced constraint has the occurrence value 1 as a start.

- Reactivate: The reactivate transition considers a constraint that was already added to the store before. It became re-activated and was added to the stack. The transition activates the constraint by associating it with an occurrence value starting with 1.
- Apply: This transition applies a CHR rule r if an active constraint matched a constraint in the head of r with the same occurrence number. If the matched constraint is part of the constraints to be removed, it is also removed from the stack. Otherwise, it is kept in the constraint store and the stack.
- Drop: This transition removes the active constraint $c\#i : j$ from the stack when there no more occurrences to check. This occurs when the occurrence number of the active constraint does not appear in the program. In other words, the existing ones were tried.
- Default: This transition proceeds to the next occurrence of the constraint if the currently active one could not be matched with the associated rule. This transition ensures that all occurrences are tried.

3 CHR^{vis}: An Animation Extension for CHR

The proposed extension aims at embedding visualization and animation features into CHR programs. The basic idea is that some constraints, the interesting ones, are annotated by visual objects. Thus on adding/removing such constraints to/from the constraint store, the corresponding graphical object is added/removed to/from the graphical store. These constraints are thus treated as interesting events. Interesting constraints are those constraints that directly represent/affect the basic data structure used along the program. Visualizing such constraints thus leads to a visualization of the execution of the corresponding program. In addition, changes in the constraint store affects the data structure and its visualization. This results in an animation of the execution. For example, in a program to encode the "Sudoku" game, the interesting constraints would be those representing the different cells in the board and their values [14, 13].

The approach aims at introducing a generic animation platform independent of the implemented algorithm. This is achieved through two features. First, annotation rules are used. The idea of using interesting events for animating programs was introduced before in Balsa [5] and Zeus [4]. Both systems use the notion of interesting events. However, users need to know many details to be able to use them. CHR^{vis} eliminated the need for the user to know any details about the animation. The second feature is outsourcing the animation into an existing visual tool. For proof of concept, Jawaa [10], was used. Jawaa provides its users with a wide range of basic structures such as circle, rectangle, line, textual node , ... etc. Users can also apply actions on Jawaa objects such as movement, changing a color , ... etc. In order to define interesting events and their annotations, users are able to write their own CHR^{vis} programs with the syntax discussed later in this section. However, users are also provided with an interface (as shown in Figure 2) that allows them to specify every interesting event/constraint. In that case, the programs are automatically generated. They are then able to choose the visual object/action (from the list of Jawaa objects/actions) to link the constraint to. Once they make a choice, the panel is populated with the corresponding parameters. Parameters represent the visual properties of the object such as: color, x-coordinate, ... etc. Users have to specify a value for each parameter. A value could be one of/combinations of:

1. a constant value e.g. `100`, `blue`, ... etc.
2. the function `valueOf/1`. valueOf(X) outputs the value of the argument `X` such that `X` is one of the arguments of the interesting constraint.
3. the function `prologValue/1`. prologValue(Exp) outputs the value of the argument "`X`" computed through the mathematical expression `Exp`.
4. The keyword `random` that generates a random number.

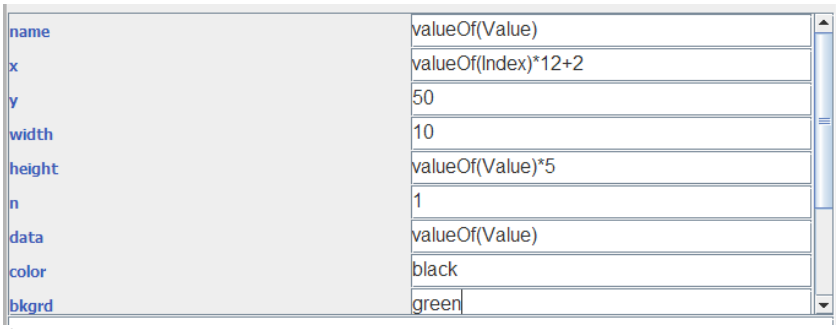


Figure 2 Annotating the cell/2 constraint

3.1 Extended Programs

This section introduces the syntax of the CHR programs that are able to produce animations on execution. In addition to the basic constructs of a CHR program, the extended version needs to specify the graphical objects to be used throughout the programs. In addition, the interesting constraints and their associations with graphical objects should be described.

3.1.1 Syntax of CHR^{vis}

The annotation rules that associate CHR constraint(s) with visual objects have the following format:
$g\ opt\_rule\_name\ @\ H_{vis}\ \Rightarrow\ Condition \mid graphical\_obj\_name\,(par_1, par_2, \ldots, par_n)\,.$
H_{vis} contains either one interesting constraint or a group of interesting constraints that are associated with a graphical object. Similar to normal CHR rules, graphical annotation rules could have a pre-condition that has to be satisfied for the rule to be applied. The literal g is added at the beginning of the rule to differentiate between CHR rules and annotation rules. A CHR^{vis} program thus has two types of rules. There are the normal CHR rules and the annotation rules responsible for associating CHR constraint(s) with graphical object(s). Moreover, there are meta-annotation rules that associate CHR rules with graphical object(s). In this case, instead of associating CHR constraint(s) with visual object(s), the association is for a CHR rule. In other words, once such rule is executed the associated visual objects are produced. The association is thus done with the execution of the rule rather than the generation of a new CHR constraint. The rule annotation is done through associating a rule with an auxiliary constraint. The auxiliary constraint has a normal constraint annotation rule with the required visual object. Such meta-annotation rule has the following format:

$$g\ opt\_rule\_name\ @\ chr\_rule\_name\ \Rightarrow\ condition \mid aux\_constraint\,(par_{1_{aux}}, \ldots, par_{m_{aux}})\,.$$

$$g\ aux\_constraint\,(par_{1_{aux}}, \ldots, par_{m_{aux}})\ \Rightarrow\ graphical\_obj\_name\,(par_1, par_2, \ldots, par_n)\,.$$

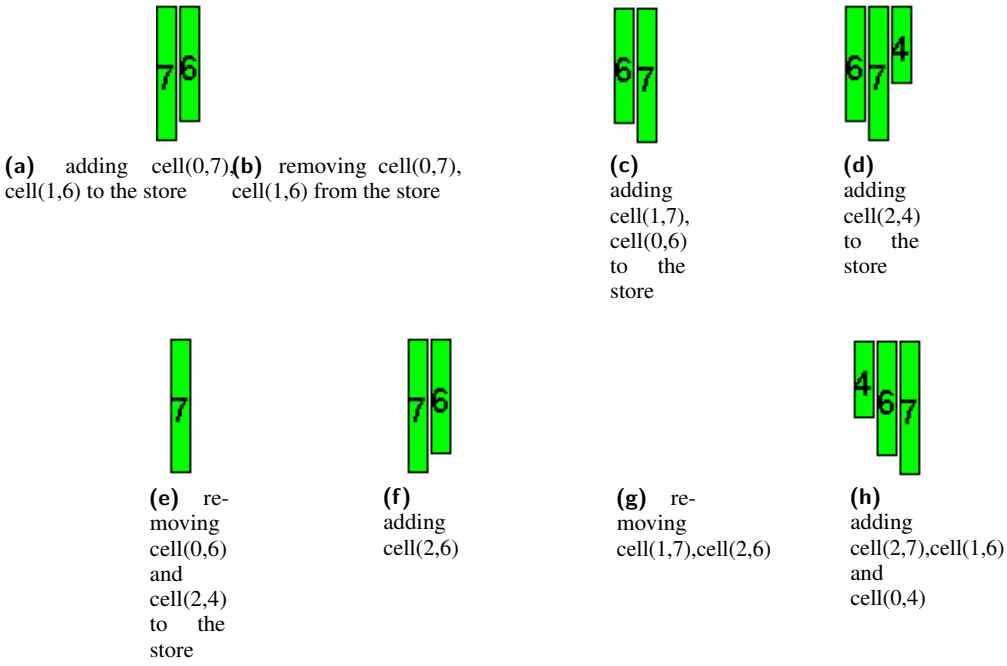
The CHR^{vis} has to determine whether head constraints affect the visualization. If this is the case, the removed gead constraints would result in removing the associated objects. In this case, head constraints should be comminicated to the tracer, Thus, a rule for `comm_head/1` has to be added to the CHR^{vis} program.
The rule `(comm_head(T) ==> T=true.)` means that head constraints are to be communicated to the tracer.
On the other hand, the rule `(comm_head(T) ==> T=false.)` means that the removed head constraints should not affect the visualization.

The program provided in Section 2 aims at sorting a list of numbers. In order to animate the execution, the elements of the list should be visualized. Changes of the elements lead to a change in the

visualization and thus animating the algorithm. The interesting constraint in this case is the `cell` constraint. As shown in Figure 2, it was associated with a rectangular node whose height is a factor of the value of the element. The x-coordinate is a factor of the index. That way, the location and size of a node represent an element of the array. The new *CHR*^{vis} program is:

```
:-chr_constraint cell/2.
:-chr_constraint comm_head/1.
comm_head(T) ==> T=true.
sort_rule @ cell(In1,V1), cell(In2,V2) <=> In1<In2,V1>V2 |
                                          cell(In2,V1), cell(In1,V2).
g ann_rule_cell @ cell(Index,Value) ==> node(valueOf(Value),
                        valueOf(Index)*12+2,
                        50,10,valueOf(Value)*5 ,1,valueOf(Value),
                        black, green, black, RECT).
```

Figure 3 shows the result of running the query `cell(0,7),cell(1,6),cell(2,4)`. As shown from the taken steps, each number added to the list and thus to the constraint store adds a corresponding rectangular node. Once `cell(0,7)` and `cell(1,6)` are added to the constraint store, the rule `sort_rule` is applicable. Thus, the two constraints are removed from the store. The rule adds `cell(1,7)` and `cell(0,6)` to the constraint store. [1] Afterwards, `cell(2,4)` is added to



(a) adding cell(0,7), cell(1,6) to the store

(b) removing cell(0,7), cell(1,6) from the store

(c) adding cell(1,7), cell(0,6) to the store

(d) adding cell(2,4) to the store

(e) removing cell(0,6) and cell(2,4) to the store

(f) adding cell(2,6)

(g) removing cell(1,7),cell(2,6)

(h) adding cell(2,7),cell(1,6) and cell(0,4)

Figure 3 Sorting an array of numbers.

the store. At this point `cell(0,6)` and `cell(2,4)` activate `sort_rule` and are removed from the constraint store. The rule first adds `cell(2,6)` to the store. At this point `cell(1,7)` and `cell(2,6)` activate `sort_rule` again. Thus they are both removed from the store. The constraints `cell(2,7), cell(1,6)` are added. Afterwards, the last constraint `cell(0,4)` is added to the store.

[1] More examples are available through `met.guc.edu.eg/chrvis/index.aspx`

As seen from Figure 3, using annotations for constraints has helped animate the execution of the sorting algorithm. However, in some of the steps, it might not have been clear which two numbers are being swapped. In that case it would be useful to use an annotation for the rule `sort_rule` instead of only annotating the constraint `cell`. The resulting program looks as follows:

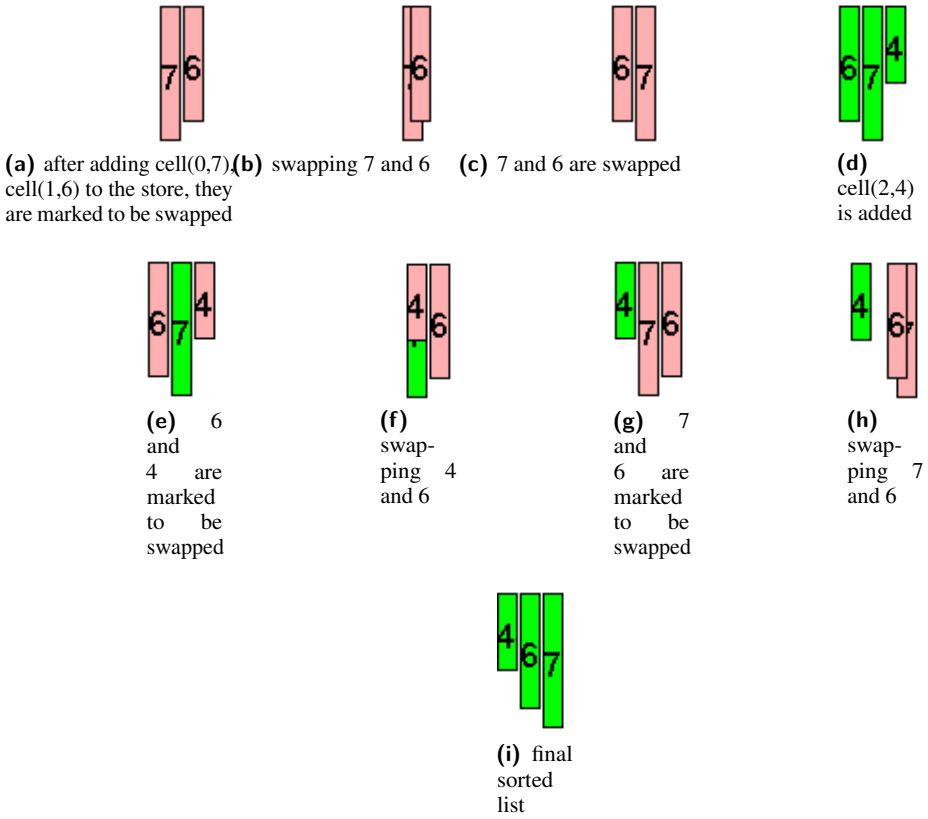
```
:-chr_constraint cell/2.
:-chr_constraint comm_head/1.

comm_head(T) ==> T=false.
sort_rule @ cell(In1,V1), cell(In2,V2) <=> In1<In2,V1>V2 |
                    cell(In2,V1), cell(In1,V2), swap(In1,V1,In2,V2).
g ann_rule_cell @ cell(Index,Value) ==> node(nodevalueOf(Value),
                          valueOf(Index)*12+2,50,10,
                          valueOf(Value)*5 , 1, valueOf(Value), black,
                          green, black, RECT).
g swap(In1,V1,In2,V2) ==> changeParam(nodevalueOf(V1),bkgrd,pink)
g swap(In1,V1,In2,V2) ==> changeParam(nodevalueOf(V2),bkgrd,pink)
g swap(In1,V1,In2,V2) ==> moveRelative(nodevalueOf(V1),
                          (valueOf(I2)-valueOf(I1))*12,0)
g swap(In1,V1,In2,V2) ==> moveRelative(nodevalueOf(V2),
                          (valueOf(I2)-valueOf(I1))*(-12),0)
g swap(In1,V1,In2,V2) ==> changeParam(nodevalueOf(V1),bkgrd,green)
g swap(In1,V1,In2,V2) ==> changeParam(nodevalueOf(V2),bkgrd,green)

g         sort_rule ==>         swap(In1,V1,In2,V2).
```

The annotations make sure that once two numbers are swapped, they are first marked with a different color (pink in this case). The two rectangular bars are then moved. The bar on the left is moved to the right. The bar on the right is moved to the left (negative displacement). The space between the start of one node and the start of the next node is 12 pixels. Thus the displacement is calculated as the difference between the two indeces multiplied by 12. After the swap is done, the two bars are colored back into green. The result of executing the query: `cell(0,7),cell(1,6),cell(2,4)` is shown in Figure 4.

3.2 Animation Formalization

The rest of the section offers a formalization of the animation to be able to run *CHR^{vis}* programs and reason about their correctness. The basic idea is introducing a new "graphical" store. *CHR^{vis}* adds, besides the classical constraint store of CHR, a new store called the graphical store. As implied by the name, the graphical store contains graphical/visual objects. Such objects are the visual mappings of the interesting constraints. Over the course of the program execution, and as a result of applying the different rules, the constraint store and the graphical store would change. As introduced before, the change of the visual objects leads to an animation of the program. The rest of the section introduces some needed definitions. It then proceeds to show the transitions of the new operational semantics.

▶ Definition 3. In *CHR^{vis}*, a state is represented by a tuple $\langle G, S, Gr, B, T, H\_ann \rangle_n$. G, S, B, T, and n have the same meanings as in a normal CHR state (goal store, CHR constraint store, built-in store, propagation history and the next available identification number) introduced in Section 2.1. Gr is a store of graphical objects. H_ann is the history of the applications of the visual annotation rules. Each element in H_ann has the following format: $\langle rule\_name, Head\_ids, Object\_ids \rangle$ where



(a) after adding cell(0,7), cell(1,6) to the store, they are marked to be swapped

(b) swapping 7 and 6

(c) 7 and 6 are swapped

(d) cell(2,4) is added

(e) 6 and 4 are marked to be swapped

(f) swapping 4 and 6

(g) 7 and 6 are marked to be swapped

(h) swapping 7 and 6

(i) final sorted list

Figure 4 Sorting an array of numbers through a rule annotation.

- *rule_name* represents the name of the fired annotation rule.
- *Head_ids* contain the ids of the head constraints that fired the annotation rule.
- *Object_ids* are the ids of the graphical objects added to the graphical store through firing *rule_name* using *Head_ids*.

▶ Definition 4. For a sequence $Sq = (c_1\#id_1, \ldots, c_n\#id_n)$, the function $get\_constraints(Sq) = (c_1 \ldots, c_n)$.

▶ Definition 5. Two sequences A and B are equivalent: $A \doteq B$ if

1. For every X, if X exists N times in A such that $N > 0$, then X exists N times in B.
2. For every Y, if Y exists N times in B such that $N > 0$, then Y exists N times in A.

▶ Definition 6.

The function $output\_graphical\_object(c(Arg_0, \ldots, Arg_n), \{Arg'_0, \ldots, Arg'_n\}, output(Object, OArg_0, \ldots, OArg_k)) = graphical\_object(Actual_0, \ldots, Actual_k)$ such that:

- $graphical\_object = Object$.
- Each parameter $Actual_n = get\_actual(OArg_n)$ such that
  - if $OArg_n$ is a constant value then $get\_actual(OArg_n) = OArg_n$.
  - if $OArg_n = valueOf(Arg_m)$ then $get\_actual(OArg_n) = (Arg'_n)$.
  - if $OArg_n = prologValue(Expr)$ then $get\_actual(OArg_n) = X$ where $Expr$ is evaluated in SWI-Prolog and binds the variable X to a value.
  - if $OArg_n = random$, then $get\_actual(OArg_n)$ is a randomly computed number.

▶ Definition 7.

The function
$generate\_new\_ann\_history(Graph\_obj, Obj\_id, rule\_name, Head\_id, H\_ann) = H'\_ann$ such that: in the case where $\langle rule\_name, Head\_id, Objects\_ids\rangle \in H\_ann$,
$H'\_ann = H\_ann - \langle rule\_name, Head\_id, Objects\_ids\rangle \cup \langle rule\_name, Head\_id, Objects\_ids \cup \{Obj\_id\}\rangle$,

▶ Definition 8.

The function $remove\_gr\_obj(G\_store, Rem\_head\_id, H\_ann) = G'\_store$ such that: in the case where there is some Tuple $T : \langle rule\_name, Head\_ids, Objects\_ids\rangle$ such that
$T \in H\_ann \wedge Rem\_head\_id \subseteq Head\_ids$.
In this case, $G'\_store = G\_store - \cup_i (Obj_i \; where \; Obj_i \in Objects\_ids)$.

▶ Definition 9.

The function $contains(H\_ann, \langle rule, Head_{ids}\rangle)$ is:

- $true$ in the case where H_ann contains a tuple of the form $\langle rule, Head_{ids}, Objects\rangle$.
- $false$ in the case where H_ann does not contain a tuple of the form $\langle rule, Head_{ids}, Objects\rangle$.

---

1. **Solve+wakeup** :
$\langle [c|A], S_0 \cup S_1, Gr, B, T, H\_ann\rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, Gr, B', T, H\_ann\rangle_n$
given that c is a built-in constraint and $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
and $wakeup(S_0 \cup S_1, c, B) = S_1$

2. **Activate**:
$\langle [c|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{activate} \langle [c\#n : 1|A], \{c\#n\} \cup S, Gr, B, T, H\_ann\rangle_{n+1}$
given that c is a CHR constraint.

3. **Reactivate**:
$\langle [c\#i|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{reactivate} \langle [c\#i : 1|A], S, Gr, B, T, H\_ann\rangle_n$
given that c is a CHR constraint.

4. **Draw**:
$\langle [\langle Obj\#\langle r, id(H)\rangle\rangle|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{draw}$
$\langle A, S, Gr \cup \{Obj\#n\}, B, T, H\_ann'\rangle_{n+1}$
given that Obj is a graphical object: $graphical\_object(Actual_0, \ldots, Actual_k)$.
and $H\_ann' = generate\_new\_ann\_history(Obj, n, r, id(H), H\_ann)$
The actual parameters of $graphical\_object$ are used to visually render the object.

5. **Update Store** :
$\langle [\langle Obj\#\langle r, id(H)\rangle\rangle|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{updatestore} \langle A, S, Gr', B, T, H\_ann\rangle_n$
given that Obj is a graphical action: $graphical\_action(Actual_0, \ldots, Actual_k)$.
$Gr' = update\_graphical\_store(Gr, graphical\_action(Actual_0, \ldots, Actual_k))$
The function $update\_graphical\_store$ uses the actual parameters of $graphical\_action$ to update the attributes of the graphical objects available in the graphical store tht are affected by the action.

6. **Apply_Annotation**:
$\langle [c\#i : j|A], H \cup S, Gr, B, T, H\_ann\rangle_n \mapsto_{apply\_annotation}$
$\langle [Obj\#\langle r, id(H)\rangle, c\#i : j|A], H \cup S, Gr, B, T, H\_ann \cup \{\langle r, id(H), \{\}\rangle\}\rangle_n$
where there is: a renamed, constraint annotation rule with variables y' of the form:
$g\ r\ @\ H' \implies Condition \mid Obj'$
where c is part of H' and

$(CT) \models \exists(B) \wedge \forall(B \implies \exists y'(chr(H) = (H') \wedge Condition \wedge output\_graphical\_object(H', y', Obj') = Obj))$ and $\neg(contains(H\_ann, (r, id(H))))$[2]
7. **Apply** :
$\langle [c\#i : j \mid A], H_k \cup H_r \cup S, Gr, B, T, H\_ann \rangle_n \mapsto_{apply}$
$\langle C + H + A, H_k \cup S, Gr, chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge G \wedge B$
$T \cup \{\langle r, id(H_k) + id(H_r) \rangle\}, H\_ann \rangle_n$ where:
• there is no applicable constraint annotation rule for c (or part of it).
(i.e. every applicable rule has already been applied).
In other words, for renamed-apart every annotation rule with variables y':
$g\ r\ @\ H' \implies Cond \mid Obj'$ where,
c is part of $H' \wedge (CT) \models \exists(B) \wedge \forall(B \implies \exists y'(chr(H) = (H') \wedge Condition))$
, it is already the case that: $(contains(H\_ann, (r, id(H)))) = true$
• There is a renamed rule in P_{vis} with the form $r\ @\ H'_k \setminus H'_r \Leftrightarrow G \mid C$.
with variables x' and the jth occurrence of c is part of the head of the renamed rule,
where $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x' ((chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge G))$
and $\langle r, id(H_k) + id(H_r) \rangle \notin T$.
If c occurs in H'_k then $H = [c\#i : j]$ otherwise $H = []$.
If the program communicates the head constraints (i.e. contains `comm_head(T) ==> T=true`) then $Gr' = remove\_gr\_obj(G, id(H_r), H\_ann)$
8. **Drop**:
$\langle [c\#i : j \mid A], S, Gr, B, T, H\_ann \rangle_n \mapsto_{drop} \langle A, S, Gr, B, T, H\_ann \rangle_n$
given that $c\#i : j$ is an occurrenced active constraint
and c has no occurrence j in the program
and that there is no applicable constraint annotation rule for the constraint c.
That could thus imply that all existing ones were tried before.
8. **Default**:
$\langle [c\#i : j \mid A], S, Gr, B, T, H\_ann \rangle_n \mapsto_{default} \langle [c\#i : j+1 \mid A], S, Gr, B, T, H\_ann \rangle_n$
in case there is no other applicable transition.

Table 2 Transitions of ω_{vis}

Table 2 shows the basic transitions of ω_{vis}. To make the transitions easier to follow, table 2 shows the transitions needed to run CHR programs with constraint annotation rules. Annotations of CHR rules are thus discarded from the set of transitions. ω_{vis} allows for running programs that contain constraint annotations. The three transitions *apply_annotation*, *draw* and *updatestore* are responsible for dealing with the graphical store and its constituents. The transition, *apply_annotation*, applies a constraint annotation rule. The rest of the transitions, such as solve, introduce and apply, have the same behavior as in ω_r. These transitions do not affect the graphical store or the application history of the annotation rules. The transitions affecting the graphical store are:

1. *Draw:* The new transition *draw* adds a graphical object (Obj) to the graphical store. Since multiple copies of a graphical object are allowed, each object is associated with a unique identifier.
2. *Update Store:* This transition applies a graphical action to the objects in the graphical store. This could thus change some of the aspects of the drawn graphical object(s).
3. *Apply_Annotation:* The *Apply_Annotation* transition applies a constraint annotation rule (ann_rule). An annotation rule is applicable if the CHR constraint store contains matching

[2] For simplicity, the annotation rule is considered to contain one graphical output object. In general, the rule could associate constraint(s) with multiple objects.

constraints. The condition of the rule has to be implied by the built in store under the matching. The built in constraint store B is also first checked for satisfiability. For the rule to be applied, it should not have appeared in the history of applied annotation rules with the same constraints i.e. it should be the first time the constraint(s) fire this annotation rule. Executing the rule adds to the goal the graphical object in the body of the executed annotation rule. The history of annotation rules is updated accordingly with the name of the rule in addition to the id(s) of the CHR constraint(s) in the head. In fact, this transition has a higher precedence than the transition *apply*. Thus in the case where an annotation rule and a CHR rule are applicable, the annotation rule is triggered first. The precedence makes sure that graphical objects are added in the intended order to ensure producing correct animations.

▶ Definition 10 (Built-In Store Equivalence). Two built-in constraint stores B_1 and B_2 are considered equivalent iff:
$(CT) \models \forall(\exists_{y1}(B_1) \leftrightarrow \exists_{y2}(B_2))$ where y_1 and y_2 are the local variables inside B_1 and B_2 respectively. The equivalence thus basically ensures that there are no contradictions in the substitutions since local variables are renamed apart in every CHR program. The equivalence check thus ensures the logical equivalence rather than the syntactical equivalence.

▶ Definition 11. A CHR^{vis} state $St_{vis} = \langle G_{vis}, S_{vis}, Gr_{vis}, B_{vis}, T_{vis}, T_{visAnn}\rangle_{n_{vis}}$ is equivalent to a CHR state $St = \langle G, S, B, T\rangle_n$ if and only if

1. $get\_constraints(G_{vis}) \doteq get\_constraints(G)$ according to Definition 5.
2. $get\_constraints(St_{vis}) \doteq get\_constraints(S) = C$ according to Definition 5.
3. B_{vis} and B are equivalent according to Definition 10.
4. $T_{vis} = T$
5. $n_{vis} \geq n$

The idea is that a CHR^{vis} state basically has an extra graphical store. The correspondence check is effectively done through the CHR constraints since they are the most distinguishing constituents of a state. Thus, the constraint store and the stack should contain the same constraints. The propagation history should be also the same indicating that the same CHR rules have been applied. n_{vis} could, however, have a value higher than n. This is due to the fact that graphical objects have identifiers. The definition of state equivalence described here follows the properties introduced in [11]. However, it is stricter.

▶ **Theorem 1** (Soundness). *Given a CHR program P (running under ω_r) along with its user defined annotations and its corresponding $P_{CHR^{vis}}$ program (running under ω_{vis}), for the same query Q, every derived state $S_{chr_{vis}}$: $Q \mapsto^*_{\omega_{vis}} S_{chr_{vis}}$ has en equivalent state S_{chr}: $Q \mapsto^*_{\omega_r} S_{chr}$*

Proof.
Base Case:
For the initial query the two states Q, $S_{chr_{vis}} = \langle Q, \{\}, \{\}\rangle$ and $S_{chr} = \langle Q, \{\}\rangle$ are equivalent according to Definition 11.
Induction Hypothesis: Suppose that there are two equivalent derived states
$S_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann\rangle_m$ and $S_{chr} = \langle A, S, B, T\rangle_n$ such that $Q \mapsto^i_{\omega_{vis}} S_{chr_{vis}}$ and $Q \mapsto^j_{\omega_r} S_{chr}$.
Induction Step:
The proof shows that any transition applicable to $S_{chr_{vis}}$ under ω_{vis} produces a state S'_{chr} such that under ω_r applying a transition to S_{chr} (which is equivalent to $S_{chr_{vis}}$) produces a state S'_{chr} that is equivalent to S_{chr}.
The different cases are enumerated below:

1. **Applying solve+wakeup to $S_{chr_{vis}}$:**
   Under ω_{vis}, solve+wakeup is applicable in the case where the stack has the form $[c|A]$ such that c is a built-in constraint and $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
   and $wakeup\,(S_0 \cup S_1, c, B) = S_1$ such that
   $S_{chr_{vis}} \mapsto_{solve+wake} S'_{chr_{vis}} : \langle S_1 + A, S_0 \cup S_1, Gr, B', T, H\_ann\rangle_m$. Since $S_{chr_{vis}}$ and S_{chr} are equivalent, S_{chr} has an equivalent stack and built-in store according to Definition 11. Thus the corresponding transition *solve+wakeup* is applicable to S_{chr} under ω_r producing a state S'_{chr} such that: $S'_{chr} = \langle S_1 + A, S_0 \cup S_1, B', T\rangle_n$. According to Definition 11, the two states $S'_{chr_{vis}}$ and S'_{chr} are equivalent.
2. **Applying Activate:**
   Such a transition is applicable to $S_{chr_{vis}}$ under ω_{vis} in the case where the top of the stack of $S_{chr_{vis}}$ contains a CHR constraint c. In this case:
   $S_{chr_{vis}} : \langle [c|A], S, Gr, B, T, H\_ann\rangle_m \mapsto_{activate} S'_{chr_{vis}} : \langle [c\#m : 1|A], \{c\#m\} \cup S, Gr, B, T, H\_ann\rangle_{m+1}$ given that c is a CHR constraint.
   The equivalent state S_{chr} has the same stack triggering the transition *Activate* under ω_r producing a state $S'_{chr} : \langle [c\#n : 1|A], \{c\#n\} \cup S, Gr, B, T, H\_ann\rangle_{n+1}$ which is also equivalent to $S'_{chr_{vis}}$
3. **Applying Reactivate:**
   In this case, $S_{chr_{vis}} \mapsto_{reactivate} S'_{chr_{vis}} \langle [c\#i : 1|A], S, Gr, B, T, H\_ann\rangle_m$
   such that $S_{chr_{vis}} = \langle [c\#i|A], S, Gr, B, T, H\_ann\rangle_m$ and c is a CHR constraint.
   The equivalent state S_{chr} has an equivalent stack triggering the transition *reactivate* under ω_r. The transition application produces $S'_{chr} : \langle [c\#i : 1|A], S, B, T\rangle_n$ which is also equivalent to $S'_{chr_{vis}}$.
4. According to Definition 11 and since $S_{chr_{vis}}$ is equivalent to S_{chr}, they both have the same stack. The transition **Draw** is only applicable if the top of the stack contains a graphical object. Since the stack of S_{chr} never contains graphical objects and since it is equivalent to $S_{chr_{vis}}$, the stack of $S_{chr_{vis}}$ at this point does not contain graphical objects as well. Thus, in this case, the transition *draw* would not be applicable to $S_{chr_{vis}}$ under ω_{vis}.
5. Similarly, according to Definition 11 and since $S_{chr_{vis}}$ is equivalent to S_{chr}, the stack of $S_{chr_{vis}}$ at this point does not contain graphical actions since both states should have the same stack. The transition *update store* is only applicable if the top of the stack contains a graphical action. Thus, similarly, at this point, the transition *update store* could not be applied to $S_{chr_{vis}}$ under ω_{vis}.
6. ***Apply Annotation Rule* Transition**
   The transition *Apply Annotation* is triggered when the stack has on top a constraint associated with an annotation rule. The constraint store should contain constraints matching the head of the annotation rule such that this rule was not fired with those constraint(s) before and the pre-condition of the annotation rule is satisfied. Thus, the rule could be associated with more than one constraint including the one on top of the stack. The constraint store should however, contain matching constraints for the rest of the constraints in the head of the annotation rule.

   $S_{chr_{vis}} \mapsto_{apply\_annotation} S'_{chr_{vis}} : \langle [Obj\#\langle r, id\,(H)\rangle|A], H \cup S, Gr, B, T, H\_ann \cup \{\langle r, id\,(H), \{\,\}\rangle\}\rangle_m$
   such that $\neg contains\,(H\_ann, \langle r, id\,(H)\rangle)$. The renamed annotation rule with variables x' is :
   $g\ r\ @\ H'\ ==>\ Condition \mid Obj'$
   $(CT) \models \exists\,(B) \wedge \forall(B \implies \exists x'((chr\,(H) =$
   $H' \wedge Cond \wedge output\_graphical\_object\,(H', x', Obj') = Obj)))$
   Either the transition *draw* or *update store* is applicable to $S'_{chr_{vis}}$. The output is $S''_{chr_{vis}} : \langle A, S, Gr',$
   $T, H'\_ann\rangle_{m'}$. In case, Obj is a graphical object, then
   $H'\_ann = generate\_new\_ann\_history\,(Obj, m, r, id(H), H\_ann \cup \{\langle r, id\,(H), \{\}\rangle\}) \wedge$
   $Gr' = Gr \cup \{Obj\#m\} \wedge m' = m + 1$. In case, Obj is a graphical action, then
   $Gr' = update\_graphical\_store\,(Gr, Obj) \wedge Gr' = Gr \wedge m' = m$. Any transition applicable to $S''_{chr_{vis}}$ at this stage is covered through the rest of the cases. Thus the application of the transition $apply\_annotation$ is considered as not to affect the equivalence of the output state with S_{chr}.

7. **The Apply transition:**
In the case where a CHR rule is applicable to $S_{chr_{vis}}$, the transition *Apply* is triggered under ω_{vis}. A CHR rule r is applicable when there is a renamed version of the rule r with variables x': $(r\ @\ H'_k \setminus H'_r \Leftrightarrow\ g \mid C.)$ where $\langle r, id(H_k) + id(H_r)\rangle \notin T$ and $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'($ $chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g))$. In this case, $S_{chr_{vis}}$ has the form: $\langle [c\#i : j|G], H_k \cup H_r \cup S, Gr, B, T, H\_ann\rangle_m$. The output state $S'_{chr_{vis}}$ has the form $\langle C + H + G, H_k \cup S, Gr, B \wedge chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g, T \cup \{\langle r, id(H_k) + id(H_r)\rangle\}, H\_ann\rangle_m$. Due to the fact that S_{chr} is equivalent to $S_{chr_{vis}}$, it has the following form: $\langle [c\#i : j|G], H_k \cup H_r \cup S, B, T\rangle_n$. For the same program, the CHR rule r is applicable producing S'_{chr}: $\langle C + +H + +G, H_k \cup S, chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g \wedge B,\ T \cup \{\langle r, id(H_k) + id(H_r)\rangle_n\}$

$$H = \begin{cases} [c\#i : j] & \text{if } c \text{ occurs in } H'_k \\ [\,] & \text{otherwise} \end{cases}$$

We assume, without loss of generality, that the same renaming variables are used in both cases. Due to the fact that the same CHR rule is applied for both states, the new built-in stores are equivalent according to Definition 10. This is due to the fact that since the original states have equivalent constraint stores, we assume without loss of generality that the matchings in both cases are the same since the same rule was applied. Thus, the rule in the two programs P_{chr} and $P_{chr_{vis}}$ are renamed similarly. Since no annotation rule could be applied to a non-occurrenced constraint and according to Definition 11, the two states are equivalent.

8. **Applying Drop**
In the case where $S_{chr_{vis}} = \langle [c\#i : j|A], S, Gr, B, T, H\_ann\rangle_m$ such that c has no occurrence j in the program and case 5 is not applicable, the transition *Drop* is triggered. *Drop* produces the state $S'_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann\rangle_m$
Since S_{chr} is equivalent to $S_{chr_{vis}}$, they both have the same stack $[c\#i : j|A]$. Thus under ω_{vis}, the same transition *drop* is triggered producing $S'_{chr} : \langle A, S, B, T\rangle_n$. According to Definition 11, $S'_{chr_{vis}}$ and S'_{chr} are equivalent as well.

9. **Applying Default**
In the case where none of the above cases hold, the transition *Default* transforms $S_{chr_{vis}}$ to $S'_{chr_{vis}} : \langle [c\#i : j+1|A], S, Gr, B, T, H\_ann\rangle_m$. Similarly the equivalent state S_{chr} triggers the same transition *Default* in this case. The output state $S'_{chr} : [c\#i : j+1|A], S, B, T\rangle_n$ is still equivalent to $S'_{chr_{vis}}$

Thus in all cases an equivalent state is produced under ω_r ◀

▶ **Theorem 2** (Completeness). *Given a CHR program P (running under ω_r) along with its user defined annotations and its corresponding $P_{CHR^{vis}}$ (running under ω_{vis}) program, for the same query Q, every derived state S_{chr}: $Q \mapsto^*_{\omega_r} S_{chr}$ has an equivalent state $S_{chr_{vis}}$: $Q \mapsto^*_{\omega_{vis}} S_{chr_{vis}}$.*

For space limitations, the proof is given in B.

4 Conclusions

In conclusion, the paper presented a formalization for embedding animation features into CHR programs. The new extension, *CHR^{vis}* is able to allow for dynamic associations of constraints and rules with visual objects. The annotation rules are thus activated on the execution of the program to produce algorithm animations. Although the idea of using interesting events was introduced in earlier work, it was (to the best knowledge of the authors) never formalized before. In fact, no operational semantics for animation was proposed before. The paper offered operational semantics for *CHR^{vis}*. It

thus provides a foundation for formalizing the animation process in general and for CHR programs in particular. In the future, with the availability of formal foundations through ω_{vis}, the possibility of using CHR^{vis} as the base of a pure a visual representation for CHR should be investigated. A group of students in the German University in Cairo were exposed to the classic textual tracer and the new visual racing facility in a focus group. Most of the students stated that for them it was hard to use the textual trace to understand how a program works. They preferred to see the visual tracer which according to a conducted survey helped them understand what the presented CHR programs do.

References

1. Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *In Proceedings of 11th Workshop on Logic Programming Environments*, 2001.
2. Slim Abdennadher and Nada Sharaf. Visualization of CHR through Source-to-Source Transformation. In Agostino Dovier and Vítor Santos Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 109–118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
3. Hariolf Betz, Frank Raiser, and Thom Frühwirth. A Complete and Terminating Execution Model for Constraint Handling Rules. *Theory and Practice of Logic Programming*, 10:597–610, 7 2010.
4. Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *VL*, pages 4–9, 1991.
5. Marc H. Brown and Robert Sedgewick. A System for Algorithm Animation. In *Proceedings of*, SIGGRAPH '84, pages 177–186, New York, NY, USA, 1984. ACM.
6. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
7. Thom Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.
8. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.
9. Thom W. Frühwirth. Constraint handling rules - what else? In Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman, editors, *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, volume 9202 of *Lecture Notes in Computer Science*, pages 13–34. Springer, 2015.
10. Willard C. Pierson and Susan H. Rodger. Web-based Animation of Data structures using JAWAA. In John Lewis, Jane Prey, Daniel Joyce, and John Impagliazzo, editors, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, pages 267–271. ACM, 1998.
11. GFrank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of chr states revisited. In *6th International Workshop on Constraint Handling Rules (CHR)*, pages 34–48, 2009.
12. Matthias Schmauss. An Implementation of CHR in Java. Master's thesis, Master Thesis, Institute of Computer Science, LMU, Munich, Germany, November 1999.
13. Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. CHRAnimation: An Animation Tool for Constraint Handling Rules. In Maurizio Proietti and Hirohisa Seki, editors, *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, volume 8981 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2014.
14. Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. Visualization of Constraint Handling Rules. *CoRR*, abs/1405.3793, 2014.
15. John Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, May 1992.

A CHR^{vis} to CHR^{r} Transformation Approach

The aim of the transformation is to eliminate the need of doing any compiler modifications in order to animate CHR programs. A CHR^{vis} program P^{vis} is thus transformed to a corresponding CHR^{r} program P with the same behavior. P is thus able to produce the same states in terms of CHR constraints and visual objects as well. A similar transformation was introduced in [13].

As a first step, the transformation adds for every constraint `constraint/n` a rule of the form:
$comm\_cons\_constraint \ @ \ constraint\,(X_1,X_2,...,X_n) \ \Rightarrow \ check\,(status, false) \ |$
$communicate\_constraint\,(constraint\,(X_1,X_2,...,X_n))\,.$

The extra rule ensures that every time a `constraint` is added to the store, the tracer (*external module*) is notified. If `constraint` was annotated as an interesting constraint, its corresponding annotation rule is activated producing the corresponding visual object(s). The new rules communicate any `constraint` added to the constraint store.

The user can also choose to communicate to the tracer the head constraints since they could affect the animation. A removed head constraint could affect the visualization in case it is an interesting constraint. In this case, if the user chose to communicate head constraints, the associated visual object, produced before, should be removed from the visual trace.[3].

As a second step, the transformer adds for every compound constraint-annotation of the form: $cons_1,\ldots,cons_n \ ==> \ annotation\_constraint_{cons_1,\ldots,cons_n}\,(Arg1,\ldots,Arg_m)$, a new rule of the form: $compound_{cons_1,\ldots,cons_n} \ @ \ cons_1\left(Arg_{cons_{1_1}},\ldots,Arg_{cons_{1_{1x}}}\right),\ldots,cons_n\left(Arg_{cons_{n_1}},\ldots,Arg_{cons_{n_{ny}}}\right)$
$\Rightarrow check\,(status, false) \ | \ annotation\_constraint_{cons_1,\ldots,cons_n}\,(Arg1,\ldots,Arg_m)$.

By default, a propagation rule is produced to keep $cons_1,\ldots,cons_n$ in the constraint store. However, the transformer could be instructed to produce a simplification rule instead. The annotation is triggered whenever $cons_1,\ldots,cons_n$ exist in the constraint store. Whenever this is the case, the rule $compound_{cons_1,\ldots,cons_n}$ is triggered producing the annotation constraint. Since the annotation constraint is a normal CHR constraint, it is automatically communicated to the tracer using the previous step.

As a third step, the CHR rules annotated by the user as interesting rules should be transformed. The idea is that the CHR constraints produced by such rules should be ignored. In other words, even if the rule produces an interesting CHR constraint, it should not trigger the corresponding constraint annotation. Instead, the rule annotation is triggered.

Hence, to avoid having problems with this case, a generic *status* is used throughout the transformed program P_{Trans}. Any rule annotated by the user as an interesting rule changes the `status` to *true* at execution. However, the rules added in the previous two steps check that the status is set to $false$. In other words, if the interesting rule is triggered, no constraint is communicated to the tracer since the guard of the corresponding *communicate_constraint* rule fails. Any rule $rule_i @ H_K \setminus H_R \Leftrightarrow G \mid B$ with the corresponding annotation $rule_i \ ==> \ annotation\_constraint_{rule_i}$ is transformed to: $rule_i @ H_K \setminus H_R \Leftrightarrow G \mid set\,(status,true)\,,\ B,\ annotation\_constraint_{rule_i}$,
$set\,(status, false)$. In addition, the transformer adds the following rule to P_{Trans}:
$comm\_cons_{annotation\_constraint_{rule_i}} \ @ \ annotation\_constraint_{rule_i} \Leftrightarrow$
$communicate\_constraint\left(annotation\_constraint_{rule_i}\right).$
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.

The aim of the transformation process is to produce a CHR^{r} program (P_{trans}) that is able to

[3] 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.

perform the same behavior of the corresponding CHR^{vis} program (P_{vis}) which basically contains the original CHR program P along with the constraint(s) and rule annotations. This section shows that the transformed program, using the steps shown previously, is a correct one. In other words, for the same query Q, P_{trans} produces an equivalent state to the one produced by P. As seen from the previous section ω_{vis} was proven to be sound and complete. This implies that any state reachable by ω_r is also reachable by ω_{vis}. In addition, any state reachable by ω_{vis} is also reachable by ω_r. The focus of this section is the initial CHR program provided by the user. The aim is to make sure that P_{trans} produces the same CHR constraints that P produces to make sure that the transformation did not change the behavior that was initially intended by the programmer. The focus is thus to compare how P and P_{trans} perform over ω_r.

B Completeness Proof

Proof.
Base Case: For a given query Q, the initial state in ω_r is $S_{chr} = \langle Q, \{\}, \{\}, \{\}\rangle_1$. The initial state in ω_{vis} is $S_{chr_{vis}} = \langle Q, \{\}, \{\}, \{\}, \{\}, \{\}\rangle_1$. [4] According to Definition 11 S_{chr} and $S_{chr_{vis}}$ are equivalent.

Induction Hypothesis: Suppose that there are two equivalent derived states $S_{chr} = \langle A, S, B, T\rangle_n$ and $S_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann\rangle_m$ such that $Q \mapsto^i_{\omega_r} S_{chr}$ and $Q \mapsto^j_{\omega_{vis}} S_{chr_{vis}}$.

Induction Step: According to the induction hypothesis, S_{chr} and $S_{chr_{vis}}$ are equivalent. The rest of the proof shows that any transition applicable to S_{chr} in ω_r produces a state that has an equivalent state produced by applying a transition to $S_{chr_{vis}}$ in ω_{vis}. Thus, no matter how many times the step is repeated, the output states are equivalent.

- **Applying solve+wakeup:**
  In this case, $S_{chr} \mapsto S'_{chr}$ such that:
  $S_{chr} : \langle [c|A], S_0 \cup S_1, B, T\rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, B', T\rangle_n$
  Transition *solve+wakeup* is applicable if:

  1. c is a built-in constraint
  2. $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
  3. $wakeup(S_0 \cup S_1, c, B) = S_1$

  $S_{chr_{vis}}(\langle Stack, S_{chr_{vis}}, Gr, B_{vis}, T_{vis}, T_{ann}\rangle_m)$ is equivalent to $S_{chr}(\langle [c|A], S_0 \cup S_1, B, T\rangle_n)$. Thus according to Definition 11, $Stack = [c|A] \wedge S_{chr_{vis}} = S_0 \cup S_1 \wedge B_{vis} = B \wedge T_{vis} = T \wedge m \geq n$. Thus accordingly, the transition $solve + wakeup$ is applicable to $S_{chr_{vis}}$ under ω_{vis} producing $S'_{chr_{vis}}$:$\langle S_1 + A, S_0 \cup S_1, Gr, B \wedge c, T, H\_ann\rangle_m$. According to Definition 11, S'_{vis} is equivalent to S'_{chr}

- **Applying Activate:**
  In this case, $S_{chr} = \langle [c|A], S, B, T\rangle_n$ where c is a CHR constraint. Thus $S_{chr} \mapsto_{activate} S'_{chr}$: $\langle [c\#n : 1|A], c\#n \cup S, B, T\rangle_{n+1}$.
  Since $S_{chr_{vis}}(\langle Stack, S_{chr_{vis}}, Gr, B_{vis}, T_{vis}, T_{ann}\rangle_m)$ is equivalent to $S_{chr}(\langle [c|A], S_0 \cup S_1, B, T\rangle_n)$. Thus according to Definition 11: $Stack = [c|A] \wedge S_{chr_{vis}} = S \wedge B_{vis} = B \wedge T_{vis} = T \wedge m \geq n$
  Accordingly, $S_{chr_{vis}} \mapsto_{activate} S'_{chr_{vis}} : \langle [c\#m : 1|A], \{c\#m\} \cup S, Gr, B, T, T_{ann}\rangle_{m+1}$ which is equivalent to S'_{chr}. (Since $m \geq n$, then $m + 1 \geq n + 1$).

[4] Throughout the different proofs, identifiers are omitted for brevity

- **Applying Reactivate:**
  The transition *reactivate* is applicable if the stack has on top of it an element of the form $c\#i$ where c is a CHR constraint. In this case $S_{chr} = \langle [c\#i|A], S, B, T\rangle_n$. Accordingly, $S_{chr} \mapsto_{reactivate} S'_{chr} : \langle [c\#i : 1|A], S, B, T\rangle_n$. Since $S_{chr_{vis}}$ and S_{chr} are equivalent, then $S_{chr_{vis}}$ has the same stack. $S_{chr_{vis}} = \langle [c\#i|A], S, Gr, B, T, T_{ann}\rangle_m$ triggers the transition reactivate producing $S'_{chr_{vis}} : \langle [c\#i : 1|A], S, Gr, B, T, T_{ann}\rangle_m$ which is also equivalent to S'_{chr}. Since c is not associated with an occurrence yet, no annotation rule is applicable at this point.

- **Applying the transition Apply**
  The transition *Apply* is triggered under ω_r in the case where $S_{chr} = \langle [c\#i : j|A], H_1 \cup H_2 \cup S, B, T\rangle_n$ such that the jth occurrence of c is part of the head of the re-named apart rule with variables x':
  $r \ @ \ H'_1 \setminus H'_2 \ \Leftrightarrow \ g \mid C$.
  such that:
  $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'(chr(H_1) = (H'_1) \wedge chr(H_2) = (H'_2) \wedge g)))$ and $\langle r, id(H_1) + id(H_2)\rangle \notin T$.
  Thus in such a case $S_{chr} \mapsto_{apply\ r} S'_{chr} : \langle C + H + A, H_1 \cup S, chr(H_1) = (H'_1) \wedge chr(H_2) = (H'_2) \wedge g \wedge B, T \cup \{\langle r, id(H_1) + id(H_2)\rangle\}\rangle_n$

$$H = \begin{cases} [c\#i : j] & \text{if } c \text{ occurs in } H'_1 \\ [\,] & \text{otherwise} \end{cases}$$

  Due to the fact that S_{chr} and $S_{chr_{vis}}$ are equivalent, in the case where S_{chr} triggers the transition *Apply* under ω_r, the same rule is also applicable under ω_{vis} to $S_{chr_{vis}}$. However for $S_{chr_{vis}}$, one of two possibilities could happen:

  1. There is no applicable constraint annotation rule:
     This could be due to the fact that any applicable annotation rule was already executed or that there are no applicable annotation rules at this point. In this case, the transition $apply$ is triggered right away under ω_{vis} producing a state
     $(S'_{chr_{vis}} : \langle C + H + A, H_1 \cup S, Gr, chr(H_1) = H'_1 \wedge chr(H_2) = H'_2 \wedge g \wedge B,$
     $T \cup \{\langle r, id(H_1) + id(H_2)\rangle, H\_ann\}\rangle_m)$ equivalent to (S'_{chr}). The original states are equivalent and the same rule is applied in both cases. We can assume that, without loss of generality , in the chr_{vis} program, the rule is renamed using the same variables x' resulting in the same matching. This is because the same matching should happen to be able to apply the same rule using the given constraint stores.
  2. There is an applicable annotation rule:
     In this case an annotation rule (r_{ann}) for c is applicable such that:
     $S_{chr_{vis}}\langle [c\#i : j|A], H_1 \cup H_2 \cup S, Gr, B, T, H\_ann\rangle_m \mapsto_{apply\_annotation}$
     $S'_{chr_{vis}} : \langle [Obj\#\langle r, id(H)\rangle, c\#i : j|A], H_1 \cup H_2 \cup S, Gr, B, T, H\_ann \cup \{\langle r_{ann}, id(H), \{\,\}\rangle\}\rangle_m$ according to the previously mentioned conditions.
     At this point either the transition *draw* or *update store* is applicable such that:
     $S'_{chr_{vis}} \mapsto_{draw/updatestore} S''_{chr_{vis}} : \langle [c\#i : j|A], H_1 \cup H_2 \cup S, Gr', B, T, H'\_ann\rangle_{m'}$
     In case Obj is a graphical object, the transition *draw* is applied such that: $Gr' = Gr \cup \{Obj\#m\} \ \wedge \ m' = m + 1 \ \wedge \ H'\_ann = generate\_new\_ann\_history(Obj, m, r, id(H), H\_ann \cup \{\langle r_{ann}, id(H), \{\,\}\rangle\})$.
     In case, Obj is a graphical action, the transition *update store* is applied such that:
     $Gr' = update\_graphical\_store(Gr, Obj) \wedge m' = m \wedge H'\_ann = H\_ann \cup \{\langle r_{ann}, id(H), \{\,\}\rangle\}$
     Since the two transitions, could only change the graphical stores, annotation history and the next available identifier, the equivalence of the states is not affected.

At this point ω_{vis} fires the transition $Apply$ for the same CHR rule that triggered the same transition under ω_r earlier. The produced state $S'''_{chr_{vis}}$ has the format:
$\langle C+H+A, H_1 \cup S, Gr', chr\,(H_1) = H'_1 \wedge chr\,(H_2) = H'_2 \wedge B, T \cup \{\langle r, id\,(H_1)+id\,(H_2)\rangle\},$
$H'\_ann\rangle_{m'}$. Similarly the same matching (local variable renaming x') has to be applied for the rule to fire.
Consequently, according to Definition 11, the state $S'''_{chr_{vis}}$ is still equivalent to S'_{chr}

- **Applying the transition drop:**
In the case where the top of the stack has an occurrenced active constraint $c\#i : j$ such that c has no occurrence j in the program, the transition drop is applied. Thus, $S_{chr} : \langle [c\#i : j|A], S, B, T\rangle_n \mapsto_{drop}$
$S'_{chr} : \langle A, S, B, T\rangle_n$
Since $S_{chr_{vis}}$ and S_{chr} are equivalent, the stack of both states have to be equivalent.
Thus $S_{chr_{vis}} = \langle [c\#i : j|A], S, Gr, B, T, H\_ann\rangle_m$. For ω_{vis} one of two possibilities is applicable:

1. No annotation rule is applicable. This could be either because c is not associated with any visual annotation rules or because all such rules have been already applied. In this case $S_{chr_{vis}} : \langle [c\#i : j|A], S, Gr, B, T, H\_ann\rangle_m \mapsto_{drop} S'_{CHR_{vis}} : \langle A, S, Gr, B, T, H\_ann\rangle_m$ which is equivalent to S'_{chr}.
2. The second possibility is the existence of an applicable annotation rule: transforming $S_{chr_{vis}}$ to $S'_{chr_{vis}} : \langle [Obj\#\langle r, id\,(H)\rangle, c\#i : j|A], S, Gr, B, T, H'\_ann\rangle_m$. At that point either *draw* or *update store* are to be applied transforming $S'_{chr_{vis}}$ to $S''_{chr_{vis}} : \langle [c\#i : j|A]$
$, S, Gr', B, T, H''\_ann\rangle_{m'}$. At that point, the transition drop is applicable converting $S''_{chr_{vis}}$ to $S'''_{chr_{vis}}$:
$\langle A, S, Gr', B, T, H''\_ann\rangle_{m'}$. $S'''_{chr_{vis}}$ is equivalent to S'_{chr}

- **Applying the default transition**
If none of the previous cases is applicable, $S_{chr} : \langle [c\#i : j|A], S, B, T\rangle_n \mapsto_{default}$
$S'_{chr} : \langle [c\#i : j+1|A], S, B, T\rangle_n$.
For the equivalent $S_{chr_{vis}}$, one of two possible cases could happen:

1. **Apply annotation is not applicable:**
In that case, the *Default* transition is directly applied transforming $S_{chr_{vis}} to S'_{chr_{vis}}$ such that $\langle [c\#i : j|A], S, Gr, B, T, H\_ann\rangle_m \mapsto_{default} \langle [c\#i : j+1|A], S, Gr, B, T, H\_ann\rangle_m$.
The produced state ($S'_{chr_{vis}}$) is equivalent to S'_{chr} as well.
2. **Apply annotation is applicable:**
In this case an annotation rule for one of the existing constraints is applicable such that:
$S_{chr_{vis}} \langle [c\#i : j|A], S, Gr, B, T, H\_ann\rangle_m \mapsto_{apply\_annotation}$
$S'_{chr_{vis}} : \langle [Obj\#\langle r, id(H)\rangle, c\#i : j|A], S, Gr, B,\ T, H'\_ann\rangle_m$ according to the previously mentioned conditions.
At this point, either the transition *draw* or the transition *update store* is applicable such that:
$S'_{chr_{vis}} \mapsto_{draw} S''_{chr_{vis}} : \langle [c\#i : j|A], S, Gr', B,\ T, H''\_ann\rangle_{m'}$
$S''_{chr_{vis}}$ is still equivalent to S_{chr}.
At the point where the transition *apply_annotation* is no longer applicable, the only applicable transition is *Default* transforming $S''_{chr_{vis}}$ to $S'''_{chr_{vis}}$ such that $S'''_{chr_{vis}} = \langle [c\#i : j+1|A], S, Gr', B,$
$T, H''\_ann\rangle_{m'}$. According to Definition 11, $S'''_{chr_{vis}}$ is equivalent to S'_{chr}

Thus in all cases an equivalent state is produced under ω_{vis} ◀

◀