DiagrammaticCHR: A Diagrammatic Representation of CHR Programs

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

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

***Abstract*—Recently, a new approach for embedding visualization features into Constraint Handling Rules (CHR) programs has been proposed. It allows CHR programmers to animate and visualize different algorithms implemented in CHR. Such features have become essential with CHR being a general purpose language. In this paper, a new diagrammatic representation for CHR programs is presented. The representation is able to account for the newly embedded visual features.**

***Keywords*-Visual Language, Constraint Handling Rules, Animation**

1 Introduction

Constraint Handling Rules (CHR) was initially introduced as a language for writing constraint solvers. Over the years, it has, however, been used with various applications and fields. CHR has many implementations including Java and C. However, the Prolog implementation is the most prominent one. A problem that always faced CHR programmers is debugging and tracing. With the use of CHR with various applications, a visual tracing tool was required. Such tool should not only show users which rules are being executed, the visualization should be related to the algorithm implemented by the program. Users should be shown a visual representation of the data structures used in the algorithm and how they change from one point of execution to another. This would result in an animation for the algorithm. Such animations have different advantages. They could be used by programmers for tracing. They could also be useful for beginners to the language. In addition, they could be used in classrooms for learning. In general, visualization was found to increase the effectiveness of learning [5]. Previous work[12], [11] aimed at augmenting CHR programs with visualization features. The aim was to animate different algorithms implemented in CHR without having to develop a different tracing tool for every algorithm class. The idea presented was to annotate different CHR constraints with visual objects. This would, on adding a constraint to the store, add its associated visual object to the visualization. Changes in the store thus lead to changes in the visualized objects animating the implemented algorithm.

This paper aims at introducing a new representation for CHR, *diagrammaticCHR*. The new representation is able to account for the CHR rules. However, it also able to represent the new visual features implemented in CHR. The paper presents the basic concepts behind *DiagrammaticCHR*. The paper presents the preliminary concepts behind *DiagrammaticCHR*. The aim for *DiagrammaticCHR* is to be formalized as a new visual language to account for the animation process. Such representation could be used to conduct any needed correctness proofs. The paper is organized as follows: Section 2 introduces CHR. Section 3 introduces *diagrammaticCHR*. We finally conclude with directions to future work.

2 Constraint Handling Rules

Constraint Handling Rules (CHR) introduced in [3], [4] is a rule-based declarative language. It was developed for writing constraint solvers. A CHR program consists of rules that rewrite the constraints available in the constraint store until a fixed point is reached. In this section, the syntax and semantics of CHR are introduced by example. The following single-line program can determine the minimum number among a set of numbers. Each number `A` is represented by a constraint `min(A)`.

```
find_min @ min(A), min(B) <=> A<B | min(A).
```

The rule called `find_min` operates on two constraints `min(A)` and `min(B)` (also referred to as the head of the rule). The pre-condition (also referred to as the guard of the rule) of applying the rule is that `A` is less than `B`. The rule removes the two constraints `min(A)` and `min(B)` from the store and adds a new constraint `min(A)` which is the body of `find_min`. Thus, as a result of successively applying the rule, only the smaller number remains in the store. For the query `min(8),min(5),min(0)`, the rule first compares `8` and `5` and as a result `min(8)` is removed from the store. `min(5)` is removed and then is re-added through the body of the rule. The rule is then applied on `min(5)` and `min(0)` keeping a new copy of `min(0)` and removing `min(5)`. The program was thus able to find the minimum number. `find_min` is a simplification rule. There is another type of CHR rules which is propagation rules. A propagation rule adds the constraints in the body to the store without removing anything. The head constraints are thus kept in the store. An example of propagation rules is `add_path`. This rule operates on two `path`

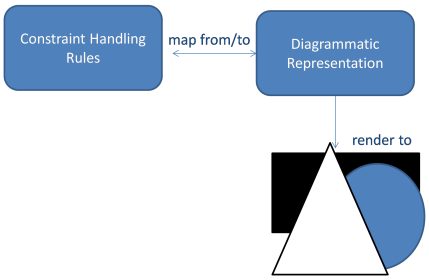

Figure 1. General View of how diagrammaticCHR is used.

constraints. A path from node `X` to node `Y` is represented by a constraint `path(X,Y)`. The rule `add_path` adds for every two paths `path(A,B)`, `path(B,C)` a new constraint `path(A,C)`. This represents the fact that if there is a path from node `A` to node `B` and a path from node `B` to node `C` then there is a path from node `A` to node `C`. In this case a new piece of information is added to the store without removing anything.

```
add_path @ path(A,B), path(B,C)
                          ==> path(A,C).
```

The third type of rules is simpagation rules which remove part of the head constraints (after the backslash symbol) and keep the other part. An example of such rule is `find_min_simp` which on application removes `min(B)` from the store and keeps `min(A)`.

```
find_min_simp @  min(A)\min(B)
                          <=> A<B | true.
```

For the query `min(8),min(5),min(0)`, the rule compares `8` and `5` and as a result `min(8)` is removed from the store, keeping only `min(5)` (without having to re-add it as in `find_min`). When applied on `min(5)` and `min(0)`, `min(5)` is removed keeping only `min(0)` in the store. `find_min_simp` is thus also able to find the minimum number.

3 Diagrammatic CHR: Syntax and Semantics

The suggested approach is to have a generic diagrammatic representation of CHR programs. The representation should also account for the newly introduced constraint annotations introduced in [12] and [11]. It should be possible for this representation to be directly rendered into an actual animation (using any tool such as Jawaa [9]).

With this approach, a formal procedure of mapping a CHR program (and its annotations) into a corresponding diagram should be provided. Moreover, it should be possible to convert any diagram into a CHR program in addition to its annotations. The new diagrammatic representation does not use constraint diagrams. The work thus builds on the concept of using diagrams for representations such as in [13]. However, Diagrammatic CHR is aimed to be a standalone visual language such as [7]. The paper presents an

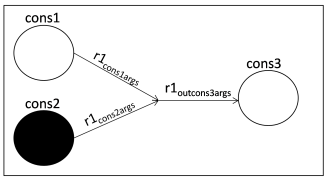

Figure 2. Basic Unit of a CHR diagram.

incremental definition of diagrammatic CHR. The available visual logic languages were not rule-based and many of them focused on mappings for first order logic and Prolog terms and relatively simple Prolog rules such as the work provided in [8], [1] and [10] Unlike the available rule-based visual languages such as [6] which are directed towards a specific application, DiagrammaticCHR offers a generic rule-based visual language. CHR had no previous visual representation. One attempt was to use Petri-nets [2]. However, the proposed mapping was die for a small subset of CHR programs.

Example 1:

Figure 2, represents the basic building block of a CHR diagram. As seen in Figure 2:

- The basic objects in the diagrams are circles.
- Each object (circle) has a label.
- diagrammaticCHR maps groups of circles (left hand side circles) into other groups (right hand side circles) using different available transitions.
- Each transition adds a group of outgoing arrows from the left hand side circles.
- Each outgoing arrow has the label $transition\_name_{circle\_labelargs}$. For example the outgoing arrow from the circle labeled `cons1` has the label $r1_{cons1args}$ for the transition `r1`.
- The transitions also add ingoing arrows to objects on the right hand side.
- Likewise, the arrows are labeled with $transition\_name_{outcircle\_labelargs}$.

Each transition can be viewed as a relation between two sets of objects (S). A transition has preconditions of application. A diagram only shows the applicable transition(s). Every time a transition is applied, a new set of objects is produced. The new set can trigger a different set of transitions leading to a new diagram with different transitions. Thus the application of a transition changes the diagram. The new diagram produced after the transition is applied is shown in Figure 3. The following could be deduced from the new diagram:

- In the diagram a new class of objects, the *rectangular objects*, are introduced.
- A dashed labeled arrow can connect a group of circles to rectangular object(s).
- The label of the dashed arrow has the form $circle\_namerectangle\_name$.

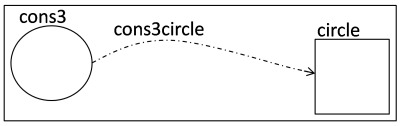

Figure 3. New state.

- Each rectangular object is labeled with a graphical object. As seen in Figure 3, a $circle$ was used.

As seen from the new diagram no more transitions are applicable so the diagram in Figure 3 is the final state.

3.1 Alphabet

The alphabet of *diagrammaticCHR* used consists of the following:

1) An infinite set of object names *O*.
2) A infinite set of transition names *Tn*.
3) A finite set of graphical objects *G*. A label for any rectangle should be one of the elements of the set G. For our case, where Jawaa is used G contains {*circle, rectangle, node, line, moveRelative, changeParam*}.
4) A infinite countable set of outgoing and ingoing solid arrow labels.
5) A infinite countable set of dashed arrow labels: $DashedLabels = O \times G$.

3.2 Syntax of Diagrammatic CHR

There are two different sets of possible arrows in a diagram: solid arrows and dotted arrows.
A solid arrow connects circular objects. The source and/or the target of a solid arrow $\mathbf{TS_{solid}} \subset \mathbf{O}$.
A dashed arrow on the other hand connects circular object(s) with rectangular (graphical) object(s).
Thus sources and/or targets of a dashed arrow $\mathbf{TS_{dashed}} \subset (\mathbf{O} \cup \mathbf{G})$.
The label of an arrow can be derived through its type (solid/dashed, ingoing/outgoing), transition name (in case of a solid arrow) in addition to the source and target.

Definition 1: A ***CHR diagram*** is a tuple: $t = < CO, Graph, Tr, Ann, SA, DA, TrCon, ParList >$ where

- $CO \subset O$ is a finite set of possible object names.
- $Graph \subseteq G$ is the set of graphical objects used throughout the different states of the diagram.
- Tr is a set of all possible transitions. Each $T \in Tr$ is considered as a relation between two sets of circular objects.For the diagram, the first set is considered as a set of input objects and the second is considered as the output objects. The inputs and outputs are represented as a tuple of $N1, N2$ elements respectively where $N1, N2$ vary from one transition to another. Therefore, $\forall_{T \in Tr} T = ((O_{in1}, ..., O_{inN1}), transition\_name, (O_{out1}, ..., O_{outN2}), (O_{in1\_status}, ..., O_{inN1\_status})) \wedge \{O_{in1}, ..., O_{inN1}\} \cup \{O_{out1}, ..., O_{outN2}\} \subset O$. The value of every element in $(O_{in1\_status}, ..., O_{inN1\_status})$ is either $true$ or $false$.
- Likewise, Ann is a set of mappings between circular and graphical objects. Every member of Ann deals with a different set of objects. Thus, $\forall_{A \in Ann} A = ((O_{in1}, ..., O_{inN1}), (Graph_{out1}, ..., Graph_{outN2})) \wedge \{O_{in1}, ..., O_{inN1}\} \subset O \wedge \{Graph_{out1}, ..., Graph_{outN2}\} \subset Graph$.
- SA is a list of solid arrows that could be derived from Tr.
- DA is a list of dashed arrows that could be derived from Ann.
- $TrCon$ is a list of transition preconditions. It is defined in more details in Definition 5.
- $ParList$ contains the parameters of all objects. $\forall_{Par \in ParList} Par = (Obj_{name}, \{Par_1, \ldots, Par_n\})$

Definition 2: A **state of a CHR diagram** is defined by the tuple $s = < CObj, CGO, AppTr, AppAnn, ActualPar >$ where:

- $CObj \subseteq CO$ is the list of available objects. Any Outgoing solid arrow $OArr \in SA$ can be only originating from some $ObjO \in CObj$. However, any ingoing solid arrow $IArr \in SA$ can be only directed towards some $ObjI \in CO, ObjI \notin CObj$.
- $CGO \subseteq Graph$ is the list of available graphical objects.
- $AppTr \subseteq Tr$ is the list of applicable transitions.
- $AppAnn \subseteq Ann$ is the list of applicable graphical objects mappings.
- $ActualPar$ is the list of acual values for all the parameters of $CObj \cup CGO$

Definition 3: Each circular/graphical object has a **parameters list**. Thus for each $Mem \in (CO \cup Graph)$ of some diagram d, there is a list Par of N different elements each representing a parameter for Mem. For example the parameter list for the graphical object $circle$ is $\{name, x, y, width, frame\_color, background\_color\}$.

Definition 4: On the other hand every $actualObj \in (CObj \cup CGO)$ of some state s has a list of **actual parameters** $\mathbf{Par\_actual}$ with the same number of elements as its corresponding Par list. However, its elements contain actual values for the corresponding parameters. For example if CGO of some s contains a $circle$ then the corresponding Par_actual could be $\{circlename, 10, 10, 30, black, green\}$.

Definition 5: A **transition precondition** $\mathbf{transprec}$ is defined as the tuple $prec = < tr\_name, objects\_conditions, input\_objects\_parameters\_conditions >$ where:

- $tr\_name \in Tn$ is the name of the transition.
- $objects\_conditions$ represents a list of conditions

that have to hold for the transition to be applied. As seen from Definition 1, every transition ($T \in Tr$) is a tuple containing in addition to the name of the transition the input and output objects. Each $obj\_con \in objects\_conditions$ is a condition between different input objects. As illustrated in Definitions 3, 4, each object has a list of parameters that is translated into actual parameters in every state of a diagram. $objects\_conditions$, represent the needed conditions between the different parameters. Such conditions thus connect parameters from different objects.

- $input\_objects\_parameters\_conditions$ represents the list of conditions on every parameter of every input object. $input\_objects\_parameters\_conditions$ is a list containing tuples of the form $< object\_name, parameter_{1condition}, ..., parameter_{Ncondition} >$. Each parameter has a corresponding condition in the tuple. If there is no specific condition about the parameter, its corresponding element in the tuple is substituted by the reserved word **none**.

Definition 6: **Outgoing Solid Arrows**
$\forall Tran, Obj_{in}(Tran \in Tr \ \wedge \ Tran = ((Objects_{input}), \ transition\_name, \ (Objects_{output}), \ (Objects_{input\_staus})) \wedge Obj_{in} \in Objects_{input}) \Rightarrow \exists SolidArrow_{Tr\_out\_Obj_{in}} (SolidArrow_{Tr\_out\_Obj_{in}} \in SA \ \wedge SolidArrow_{Tr\_out\_Obj_{in}} = transition\_name_{Obj_{in}args})$

Definition 7: **Ingoing Solid Arrows**
$\forall Tran, Obj_{out}(Tran \in Tr \ \wedge \ Tran = ((Objects_{input}), \ transition\_name, \ (Objects_{output}), \ (Objects_{input\_staus})) \wedge Obj_{out} \in Objects_{output}) \Rightarrow \exists SolidArrow_{Tr\_in\_Obj_{out}} (SolidArrow_{Tr\_in\_Obj_{out}} \in SA \ \wedge SolidArrow_{Tr \in \_Obj_{out}} = transition\_name_{outObj_{out}args})$

Definition 8: **Dashed Arrows**
$\forall Annot, \ Obj_{in}, \ Gobj_{out}(Annot \in Ann \ \wedge \ Ann = ((Obj_{in}), \ (Gobj_{out})) \Rightarrow \exists DashedArrow_{Obj_{in}\_Gobj_{out}} (DashedArrow_{Obj_{in}\_Gobj_{out}} \in SA \ \wedge \ DashedArrow_{Obj_{in}\_Gobj_{out}} = Obj_{in}Gobj_{out})$

3.3 Mapping a State to a Diagram

On successive applications of the different available transitions, the states of the CHR diagrams are modified. Thus, the actual visualized diagrams are changed. Each state corresponds to a CHR diagram. This section defines how a state is mapped to a visual CHR diagram as the ones shown in Figures 2 and 3.
For any state $S =< CObj, CGO, AppTr, AppAnn, ActualParam >$ for some diagram
$< CO, Graph, Tr, Ann, SA, DA, TrCon, ParList >$:

- A new circle is added for every $C \in CObj$. The new circle is labeled with C.
- A new rectangle is added for every $CG \in CGO$. The new rectangle is labeled with CG.
- For every $Tran_{App} \in AppTr$, any solid outgoing arrow $SolidArrow_{Tran_{App}\_out\_Obj_{in}}$ linking $Tran_{App}$ to some input circle Obj_{in} is added to the diagram. If this object has a corresponding $status$ = $false$, the circle is filled.
- For every $Tran_{App} \in AppTr$, any solid ingoing arrow $SolidArrow_{Tran_{App}\_in\_Obj_{out}}$ linking $Tran_{App}$ to some output circle Obj_{out} is added to the diagram.
- Every $Ann_{App} \in AppAnn$, adds a dashed arrow $DashedArrow_{Obj_{in}\_Gobj_{out}}$ linking the circle Obj_{in} to some rectangle $Gobj_{out}$.

4 Informal Operational Semantics

Throughout the previous sections, we showed how the constituents of the diagram are driven. The basic operation of *diagrammaticCHR* is applying a transition on a diagram and rendering a diagram into the visualization. These operations are introduced in this section.

4.1 Transition Application

Applying transitions changes the state and thus the visualized diagram.

Definition 9: In a diagram d with state s, a transition $Trans = ((O_{in1}, ..., O_{inN1}), \ transition\_name, \ (O_{out1}, ..., O_{outN2}), \ (O_{in1\_status}, ..., O_{inN1\_status}))$ is applicable iff the corresponding preconditions as stated in the $TrCon$ element of the tuple d are satisfied. This includes the following

1) There is a list of objects $NeeedObj \subseteq CObj$ of s such that the number and types of elements in $NeededObj$ can be mapped to the input objects $(O_{in1}, ..., O_{inN1})$ of $trans$.
2) The list $NeededObj$ satisfies the $objects\_conditions$ of the corresponding transition precondition tuple discussed in Definition 5.
3) The actual parameters $NeeedObj$ of parameters should satisfy the $input\_objects\_parameters\_conditions$ as explained in Definition 5.

An applicable transition
$T = (Obj_{in}, transition\_name, Obj_{out}, (Obj_{in\_status}), (Obj_{out\_par}))$
changes a state
$S_1 =< CObj_1, CGO_1, AppTr_1, AppAnn_1, ActualPar_1 >$
to
$S_2 =< CObj_2, CGO_2, AppTr_2, AppAnn_2, ActualPar_2 >$
such that

- $CObj_2$ = $(CObj_1 \cup Obj_{out})$ - any $O \in Obj_{in}$ such that $O_{status} = false$.
- $AppTr_2$, $AppAnn_2$ are computed from $ActualPar_2$ and $CObj_2$. The conditions through which a transition is determined to be applicable were discussed in Definition 9.
- $ActualPar_2$ = surviving($ActualPar_1$) $\cup$ Obj_{out_par}. surviving($ActualPar_1$) are the parameters of the objects that existed in S_1 and were not removed in S_2.

4.2 Rendering a diagram

As seen from the previous section, applying a transition changes the diagram from one state S_1 to a new state S_2. Once S_2 is produced, it is mapped to an actual diagram using the steps introduced in Section 3.3. A state should also be rendered to the actual visualization, the user sees. As seen from Definition 4, every state has a list $(CObj \cup CGO)$. Moreover every $actualObj \in (CObj \cup CGO)$ has an actual parameters list (Par_actual). Such list contains the values of the parameters needed for the visual objects (represented through a rectangle) in the diagram. The system thus adds for every $actualObj$, the corresponding graphical object using Par_actual. Such list contains the graphical related values needed to draw the corresponding object such as x-coordinate, y-coordinate, ..etc. Rendering all $actualObj$s in a state leads to showing the user the needed visualization.

4.3 Mapping with CHR

Mapping between a diagram and a CHR program should be as follows:

- CHR rules map to the transition list Tr. Guards affect the corresponding pre-conditions of the transitions.
- Visual annotations correspond to the mappings' list Ann.
- CHR constraints correspond to the different available objects CO.
- Available visual objects of the tracer (for example Jawaa) correspond to $Graph$.
- The constraint store corresponds to the $CObj$ along with its actual parameters list. As seen from Figure 2, there are shaded and white objects. A shaded object represents a constraint that is removed from the store on executing a rule. This should be also formalized through the semantics.

Using this scheme, it should be possible to formalize a mapping from CHR to diagrammatic CHR and from diagrammatic CHR to CHR.

5 Conclusion & Future Work

The paper introduced a new representation for CHR programs. The representation uses diagrams to represent the different states of a program. Through such representation, the visual features of a program were embedded. The representation facilitates the conversion from CHR to its visual state. This should in the future be used to prove that the visualized programs perform the same functionality as the original ones. This would thus prove the correctness of the visualization process (in terms of having the functionality intact). The paper only presents the preliminary ideas behind the new approach. In the future, formal operational semantics should be offered for the new diagrammatic language. In addition, an implementation should be also formalized.

References

[1] J. Agustí-Cullell, J. Puigsegur, and D. S. Robertson, "A visual syntax for logic and logic programming," *J. Vis. Lang. Comput.*, vol. 9, no. 4, pp. 399–427, 1998. [Online]. Available: http://dx.doi.org/10.1006/jvlc.1998.0090

[2] H. Betz, "Relating coloured Petri nets to Constraint Handling Rules," pp. 33–47.

[3] T. Frühwirth, "Theory and practice of constraint handling rules, special issue on constraint logic programming," *Journal of Logic Programming*, vol. 37, no. 1-3, pp. 95–138, October 1998.

[4] T. W. Frühwirth, *Constraint Handling Rules*. Cambridge University Press, 2009.

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

[6] J. J. P. Jr., "Altaaira: A rule-based visual language for small mobile robots," *J. Vis. Lang. Comput.*, vol. 9, no. 2, pp. 127–150, 1998. [Online]. Available: http://dx.doi.org/10.1006/jvlc.1998.0078

[7] M. Najork and S. M. Kaplan, "The CUBE Language," in *VL*, 1991, pp. 218–224.

[8] L. F. Pau and H. Olason, "Visual logic programming," *J. Vis. Lang. Comput.*, vol. 2, no. 1, pp. 3–15, Mar. 1991. [Online]. Available: http://dx.doi.org/10.1016/S1045-926X(05)80049-7

[9] W. C. Pierson and S. H. Rodger, "Web-based animation of data structures using jawaa," *SIGCSE Bull.*, vol. 30, no. 1, pp. 267–271, Mar. 1998. [Online]. Available: http://doi.acm.org/10.1145/274790.274310

[10] J. Puigsegur, J. Agusti, and D. Robertson, "A visual logic programming language," in *Visual Languages, 1996. Proceedings., IEEE Symposium on*, Sep 1996, pp. 214–221.

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

[12] N. Sharaf, S. Abdennadher, and T. W. Frühwirth, "Visualization of Constraint Handling Rules," *CoRR*, vol. abs/1405.3793, 2014.

[13] G. Stapleton, S. J. Thompson, A. Fish, J. Howse, and J. Taylor, "A New Language for the Visualization of Logic and Reasoning," in *DMS*, 2005, pp. 287–292.