Visualization of CHR through Source-to-Source Transformation

Slim Abdennadher and Nada Sharaf

Computer Science and Engineering Department
The German University in Cairo
`slim.abdennadher,nada.hamed@guc.edu.eg`

Abstract

In this paper, we propose an extension of Constraint Handling Rules (CHR) with different visualization features. One feature is to visualize the execution of rules applied on a list of constraints. The second feature is to represent some of the CHR constraints as objects and visualize the effect of CHR rules on them. To avoid changing the compiler, our implementation is based on source-to-source transformation.

1998 ACM Subject Classification I.2.2 Automatic Programming, D.3.2 Language Classifications

Keywords and phrases Source-to-Source Transformation, Constraint Handling Rules, Visualization

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.109

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level language especially designed for writing constraint solvers. CHR is essentially a committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved. Over the last decade, CHR has matured to a powerful general purpose language by adding several features to it. There are quite a number of implementations of CHR. The most prominent ones are in Prolog.

So far, debugging the CHR code and tracing its run in Prolog was not visualized, thus less understandable and necessitating more concentration from the CHR programmer. Additionally, it offers only a small degree of freedom for the programmer to move backwards in the trace thread. In previous work [1], a tool called VisualCHR was developed to support the development of constraint solvers written in JCHR; an implementation of CHR in Java. To implement VisualCHR, the compiler of JCHR [9] has been modified to add the feature of visualization. However, such visualization feature was not available for Prolog versions.

The aim of the paper is to introduce an approach to add visualization features for CHR (implemented in SWI-Prolog) without changing the CHR compiler. Our approach uses source-to-source transformation by providing an implementation based on the core CHR language.

The paper will discuss features to visualize the execution of CHR rules as well as the graphical representation of objects and the effect of applying rules on them without changing the compiler.

The first transformer manipulates the input programs in order to be able to visualize the execution of their rules. The provided visual tracer shows the used constraints at each step and their effect on the existing constraints. The second transformer provides the user with the possibility to choose the type of objects that represent some CHR constraints and to visualize the objects after executing the rules on the constraints.

© Slim Abdennadher and Nada Sharaf;
licensed under Creative Commons License ND

The paper is organized as follows. In Section 2, we introduce briefly the CHR language. In Section 3, some comparisons to related work will be discussed. In Section 4, the source-to-source transformation used is presented. Section 5 illustrates the visualization of the execution of the different rules and the visualization of some of the CHR constraints. Finally, we conclude with a summary and directions for future work.

2 Constraint Handling Rules by Example

A CHR rule consists of a head and a body and may contain a guard. CHR allows multiple heads and a conjunction of zero or more atoms in the guard as well as in the body of the rule. In the following, the syntax and the semantics of CHR are introduced by example.

Following is an example of a typical CHR program defining the partial order relation $\leq$ (`leq/2`). `leq(A,B)` holds if variable `A` is less than or equal to variable `B`.

```
:- use_module(library(chr)).
:- chr_constraint leq/2.

reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y) , leq(Y,X) <=> X=Y.
idempotence  @ leq(X,Y)\ leq(X,Y) <=> true.
transitivity @ leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
```

The first rule, which is called `reflexivity` (rule names are optional), is a single-headed simplification rule. It removes constraints of the form `leq(A,A)` from the constraint store. The second rule, `antisymmetry`, is a simplification rule with two heads. It replaces two symmetric `leq` constraints by an equality constraint. The equality constraint is usually handled by the host language; in this case Prolog does the unification.

Simplification rules correspond to logical equivalence, as the syntax suggests. The third rule is a simpagation rule which removes redundant copies of the same constraint. Such rules are often needed because of the multi-set semantics of CHR. Finally, the last rule (`transitivity`) is a propagation rule that adds redundant constraints. Propagation rules correspond to logical implication.

Execution proceeds by exhaustively applying the rules to a given input query. For example, given the query `leq(A,B), leq(B,C), leq(C,A)` the transitivity rule adds `leq(A,C)`. Then, by applying the `antisymmetry` rule, `leq(A,C)` and `leq(C,A)` are removed and replaced by `A=C`. Now the `antisymmetry` rule becomes applicable on the first two constraints of the original query. Now all CHR constraints are eliminated so no further rules can be applied, thus the answer `A=C, A=B` is returned.

3 Related Work

Due to the importance of source-to-source transformation and the advantages that it could bring around, various attempts have been made to incorporate and use such techniques with CHR. This section briefly mentions some of the work related to using source-to-source transformation with CHR.

In [6], a description of a source-to-source transformation technique for CHR was presented. Through the used technique, it is argued that it is rather easy to add source-to-source transformation to CHR. As discussed in [6], the input CHR program is represented in a "relational normal form" using some special CHR constraints that encode the different parts of a CHR rule such as `head/4, guard/2, body/2, pragma/2` and `constraint/1`. The transformation is then done to this form. However, a new built-in predicate should be introduced to the CHR runtime system in order to register handlers as transformers, the intended order for application and the options that could provide additional control over the expansion [6]. In [6], some applications are shown such as bootstrapping the CHR compiler. Another example extends CHR by having probabilistic choice of rules. [7] provides more details about probabilistic constraint handling rules. The whole source-to-source transformation program for probabilistic CHR has a few rules and could fit into only one page [7]. Nevertheless, the runtime system had to be extended with rules for conflict resolution [6].

Another approach to program transformation was presented in [12], namely, unfolding. In order to do this, the syntax of CHR programs has to be modified. The rules have to be annotated to be in a specific format and to have a local token store. In other words, the operational semantics ω_t [3] are replaced with some modified semantics ω_t' [12]. In general, unfolding replaces a procedure call by its definition. According to [12], the unfolding process that is performed replaces the conjunction of constraints, S (considered to be the procedure call), in the body of a rule, `r`, with the body of another rule, `v`, given that the head of the rule `v` matches S.

In [5], the specialization of rules with respect to the goal is considered which is very interesting as it optimizes the program for input values of the goal. Since rules are specialized (by modifying some of their parts), and these new rules are added, this was also considered as a step towards transformation of CHR programs.

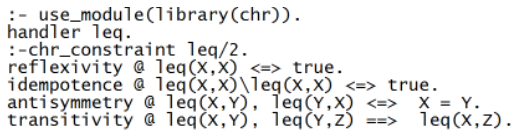
In [13], an implementation for aggregate functions in CHR was introduced. The implementation also used source-to-source transformation. However, in order to extend the current CHR systems with aggregates, a number of low-level compiler directives had to be added to the CHR system. In [11], more details about the implementation are offered. Meta CHR rules were used. Such rules rewrite the CHR rules of a specific program. A meta rule could be applied, if a single rule's head (in the original program) matched the meta occurrences of the (meta) rule. When the meta rule fires, the conjuncts of the program rule's head that caused the rule to fire are removed. The body of the meta rule could add new conjuncts to the program rule's guard or head. It could also add new rules to the program [11].

Finally, in [8], CHR^{rp}, which adds user-definable priorities to the different rules, was introduced. However, for CHR^{rp}, the priority semantics ω_p is introduced. Source-to-source transformation is used to translate CHR^{rp} into CHR.

The rest of this section goes through the differences and the advantages of the proposed system. First of all, our approach does not require any changes to the compiler or the CHR runtime system. The transformer and the output program are normal CHR programs that do not require any additions or changes to work. At the same time, the proposed system saves the user from having to translate the program into or from the "relational normal form". This translation is done automatically at the beginning through the Java application that parses the input file and finally through the transformer itself as it writes the result into a new file. Moreover, the order of the rules in the new program is the same as their order in the original program thus eliminating the rule ordering problem faced in [6]. The user is also able to control where new rules are added to the output program.

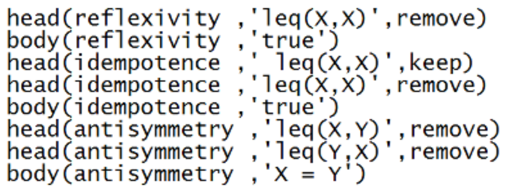
4 Source-to-Source Transformation for Visualization

In order to be able to have the required visual tracers, the original programs need to be modified to be able to interact with the tracers and produce the needed output. The



```
:- use_module(library(chr)).
handler leq.
:-chr_constraint leq/2.
reflexivity @ leq(X,X) <=> true.
idempotence @ leq(X,X)\leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=>  X = Y.
transitivity @ leq(X,Y), leq(Y,Z) ==>  leq(X,Z).
```

(a) A Sample input file.



```
head(reflexivity ,'leq(X,X)',remove)
body(reflexivity ,'true')
head(idempotence ,' leq(X,X)',keep)
head(idempotence ,'leq(X,X)',remove)
body(idempotence ,'true')
head(antisymmetry ,'leq(X,Y)',remove)
head(antisymmetry ,'leq(Y,X)',remove)
body(antisymmetry ,'X = Y')
```

(b) Some of the extracted information.

Figure 1 Sample of the information extracted through the Java application.

advantage of source-to-source transformation, in this context, is that it is able to manipulate input programs to change or add to their behavior the required functionalities without the need to do this modification manually. The proposed transformers use some of the central ideas introduced in [6]. In [6], some CHR constraints, that encode the different constituents of CHR rules, were introduced. The difference is that in the proposed transformers, instead of `head/4`, `head/3` is used since the information about the constraint's identifier is not needed. Using such CHR constraints, any CHR rule was transformed into "relational normal form".

According to [6], the modified behavior is also represented in terms of the introduced five CHR constraints. The proposed transformers, which are CHR handlers, operate on some specific CHR constraints including some of the constraints introduced in [6]. Consequently, the information regarding the different parts of the CHR rules needs to be extracted and represented in the needed format. The different parts of the rules are represented using CHR constraints. Some of them were introduced in [6] such as `guard/2 and body/2`. As mentioned before, `head/3` is used as well. For example, `head(reflexivity,'leq(X,X)',remove)`, represents the fact that the head of the rule named `reflexivity` contains `leq(X,X)` and that when executing the rule, this constraint is removed from the constraint store since `reflexivity` is a simplification rule.

Since some CHR solvers contain CHR rules as well as Prolog facts, the transformers use a new constraint named `facts/1`. This constraint is used to copy such Prolog facts into the output program. The proposed transformers write the modified program into a new file so that there is no need to alter the compiler or the CHR runtime system in anyway.

In addition to the introduced CHR constraints, there are other ones that are mainly used to write the modified rules and the facts to the output file. The proposed transformers operate on these constraints. Each of the transformers has two main functionalities. Firstly, it adds to the rules the required information and functionalities required to produce the needed visual tracer and interact with it. Secondly, it writes the new program into a file. This comes in handy since through using this arrangement, the original program does not have to be transformed every time. Instead it is transformed at the beginning and the new produced program could be used afterwards.

The question now may be how to represent the input program in the needed format so that the transformer could act on it. One possibility is to do this manually. In this case, the user would have to translate every part of the CHR program to produce the needed CHR constraints and then run the transformer on these extracted constraints. As simple as doing the translation might seem to be, this job could be very long and tedious. Therefore, a Java application is provided to parse the input file and do this translation automatically.

As shown in Figure 2, the Java application performs two tasks. First of all, it parses the input file in order to extract the needed information. In addition, it represents this information in the needed format using the new CHR constraints. Figure 1 shows some of

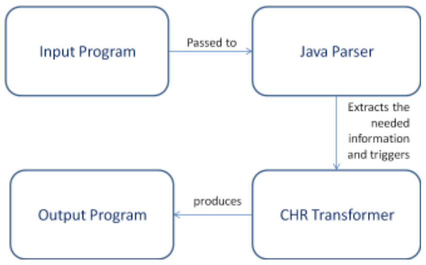


Figure 2 Overview of the system's architecture.

the extracted information from the `leq` handler presented in Section 2. Secondly, it runs the transformer using the extracted information to produce the new output program. The SWI-Prolog JPL interface [10] is used in order to run the transformer (which is a CHR program) from within the Java application. The new program has the required functionality of producing and interacting with the needed visual tracer.

The rest of the section provides an example of the output file after applying the transformation. It is concerned with the `leq` handler introduced in Section 2. The two proposed transformers produce similar output. Note that for simplicity reasons, the rules presented below are just an abstraction of the actual rules obtained by the transformers.

```
main:-initialize_visualizer_with_initial_store,proceed_tracer.
reflexivity @ leq(X,X) <=> send_visualizer_removed_head(leq(X,X)),
                           true,
                           send_visualizer_body(true),
                           rule_name(reflexivity),
                           proceed_tracer.
antisymmetry @ leq(X,Y), leq(Y,X) <=>
                            send_visualizer_removed_head(leq(X,Y)),
                            send_visualizer_removed_head(leq(Y,X)),
                            send_visualizer_body(X=Y),
                            rule_name(antisymmetry),
                            proceed_tracer,
                            X = Y.
idempotence  @ leq(X,Y)\ leq(X,Y) <=>
                            send_visualizer_removed_head(leq(X,Y)),
                            send_visualizer_kept_head(leq(X,Y)),
                            true,
                            send_visualizer_body(true),
                            rule_name(idempotence),
                            proceed_tracer.
transitivity @ leq(X,Y), leq(Y,Z) ==>
                            send_visualizer_kept_head(leq(X,Y)),
                            send_visualizer_kept_head(leq(Y,Z)),
                            send_visualizer_body(leq(X,Z)),
                            rule_name(transitivity),
                            proceed_tracer,leq(X,Z).
```

The `main` predicate rule is added to the program to be able to to initialize the visual tracer. `initialize_visualizer_with_initial_store` is an abstraction for the actions performed to initialize the tracer. The corresponding Java class is initialized (according to the applied transformer) and the initial constraints in the constraint store are sent. `proceed_tracer` is used to add a row to the tracer's tree using the initial constraints. As for the CHR rules, the corresponding data is sent to the visual tracer. Afterwards, the tracer is advanced. Similarly, advancing the tracer here means that a new row is added to the tree data structure of the tracer. The tree is not visualized unless the user presses one of the buttons of the tracer as shown in Section 5.

For example, after the `idempotence` rule is transformed, the interactions needed with the visual tracer are added to the body. `send_visualizer_removed_head(leq(X,Y))` is used to refer to the Java method call that informs the tracer that the constraint `leq(X,Y)` that appears in the head of the rule should be removed on executing the rule. On the other hand, `send_visualizer_kept_head(leq(X,Y))` is used to refer to the Java method call informing the tracer that another constraint in the rule's head is `leq(X,Y)` and that it should be kept on executing the rule. Finally, `rule_name(idempotence)` and `proceed_tracer` are also abstractions for the Java method calls that update the tracer with the rule's name and add a row to the tracer's tree data structure using all the previously sent data respectively.

5 The Visualization

As introduced before, transformation is done in order to be able to visualize the execution of the rules or to be able to visualize some of the CHR constraints. In other words, two transformers are provided. The first one enables the user to visualize the execution in a step-by-step manner. The second transformer allows the user to visualize some of the CHR constraints themselves in order to be able to visually see the result of applying the rules of a specific program on such constraints.

In order to be able to use any of the visual tracers, the initial program has to be transformed first. Afterwards, the transformed program, which is automatically saved in a new file, is consulted in the usual way. The only thing that needs to be added to the query is `main`. This is used to initialize the visualization tool (which differs according to the transformer that was used in the first step). The two visual tracers were built using Java. The interface between SWI-Prolog and Java was also done using JPL [10]. Once the goal is entered, the execution of the CHR rules proceeds in the normal way. In addition, the visual tracer window opens. This section uses the `gcd` handler:

```
:- use_module(library(chr)).
:-chr_constraint gcd/1.

r1 @ gcd(N) \ gcd(M) <=> 0<N, N=<M | L is M mod N,
                         print('added'),nl,gcd(L).
```

5.1 Visualizing The Execution

The first transformer is used in order to be able to visualize the execution of the rules. The visual tree was built using the JGraph framework [2]. In order to visualize the execution of the different rules in a step-by-step manner, the play and back buttons are used. The fast forward button allows the user to view the full tree by one click.

The program used in this section computes the greatest common divisor of two numbers. The program contains one rule named r1. There are three properties for the visualized tree.

First of all, the name of the rule that was executed appears on the edge. Secondly, the active CHR constraints are represented with green boxes, whereas the ones that were removed from the constraint store (due to a simplification or a simpagation rule) appear as red boxes. Figure 3 shows the visualization tree after visualizing only one step. As seen through the figure, the two constraints `gcd(24)` and `gcd(40)` were used by the rule named r1. As a result of applying this rule, the constraint `gcd(16)` was added to the constraint store. Now the box containing `gcd(40)` is red while the one containing `gcd(24)` is green. This means that as a result of applying the rule named r1, the constraint `gcd(40)` was removed from the constraint store while the constraint `gcd(24)` was kept. Consequently, the rule r1 is a simpagation rule. Third of all, in addition to the constraints, the tree shows all the built-in constraints and the computations using blue boxes.

5.1.1 Removing Unneeded Nodes

The visual tracer allows users to remove any unneeded nodes in order to be able to customize the tree according to their needs. For example in Figure 3 all the print statements and computations are shown. If the user does not want to see any computation, then he/she could right click on the node so that a menu with the option “Remove Similar Constraints” appear.

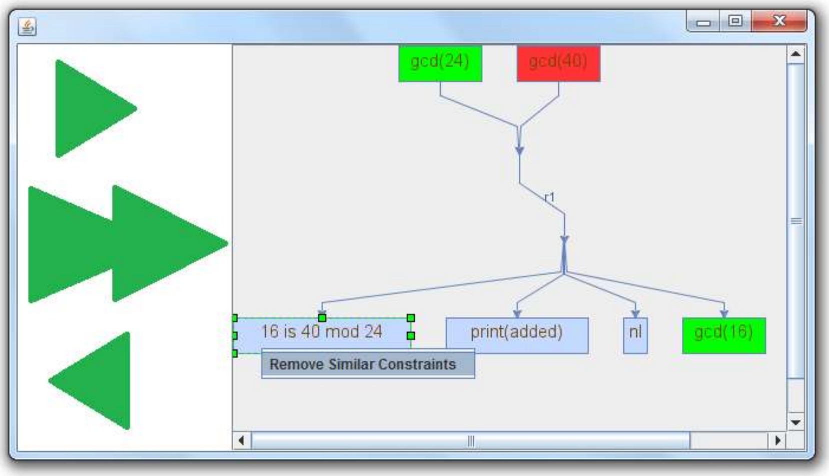


Figure 3 After the first step, right clicking a node to remove it from the tracer.

Once the user clicks on this option, the tree is redrawn without such constraints. Moreover, if the user does this in the middle of the visualization then this history is kept. In other words, if the user has chosen to remove computations then at every new step if a computation is done, it is not added to the tracer as shown in Figure 4.

5.2 Visualizing Constraints

The second transformer is used in order to be able to visualize some of the CHR constraints themselves. More specifically, if the input program manipulates a specific object, then using this transformer, the user is able to visualize the object (or the constraint itself).
For example the following program translates rectangles from one position to another.

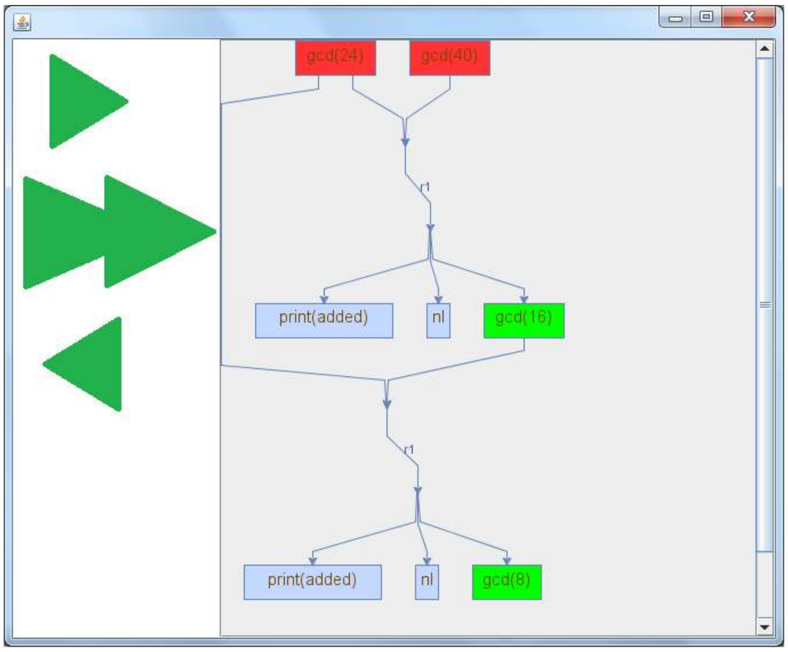


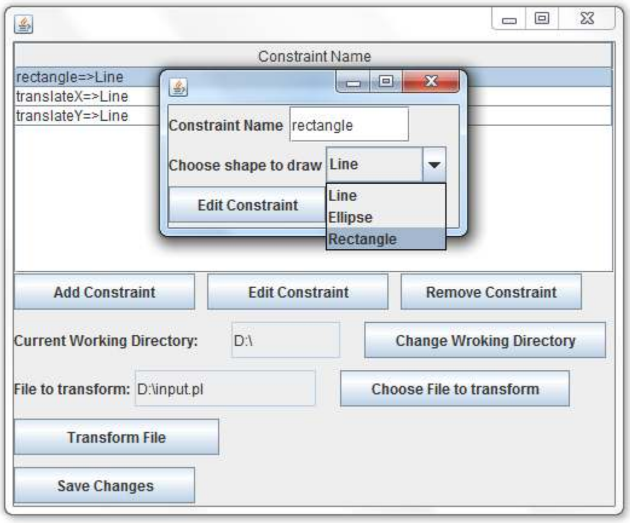
Figure 4 Node removed and history kept throughout the next step.

```
:-use_module(library(chr)).
:-chr_constraint rectangle/4,translateX/1,translateY/1.

translateX @ rectangle(X1,Y1,W,H),translateX(X) <=> NewX1 is X1+X,
                                                  rectangle(NewX1,Y1,W,H).
translateY @ rectangle(X1,Y1,W,H),translateY(Y) <=> NewY1 is Y1+Y,
                                                  rectangle(X1,NewY1,W,H).
```

In order to be able to visualize the constraints, the user has to inform the system of the CHR constraints that should be visualized and how they should be visualized. In other words, users have to specify that they want to visualize the constraint named `rectangle` and that this constraint should be visualized as a “Rectangle” object. Another example could involve the constraint `circle` that has to be visualized as an “Ellipse” object. Consequently, through the provided application, the user gets to choose the type of visualization objects that he/she would like to associate with CHR constraints. Users also decide about the constraints to visualize. They could choose to visualize all constraints, some of them or no constraints at all. Through the provided application, the user could specify all such details.

The CHR constraints in the input file appear to the user as shown in Figure 5. The default visualization object is “Line”. The user could remove the constraints that are not to be visualized such as `translateX` and `translateY` in the previous example. Users could also select a constraint and choose to edit it in order to specify the type of object that should be associated with the constraint as shown in Figure 5. Once the user is done with transforming the input file, the produced output file could be used for visualization. The result of applying each rule could be seen through clicking as shown in Figure 6. Figure 6 is the result of applying the query `rectangle(10,10,50,50), translateX(100), translateY(100)` to the resulting output file (the file produced after applying the transformation). Since the object chosen for visualizing the `rectangle` constraint is “Rectangle”, the arguments `10, 10, 50`



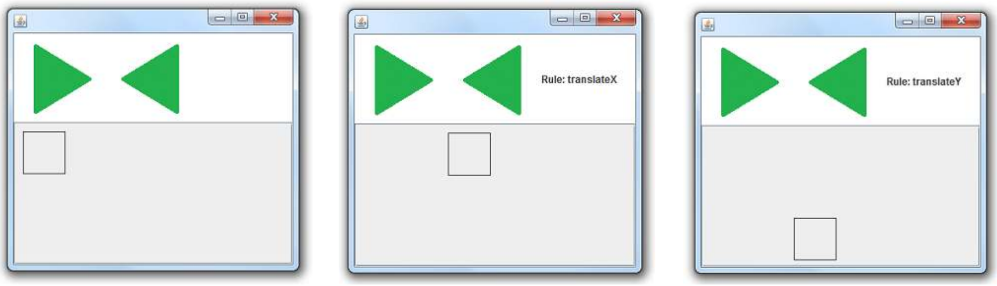
Constraint Name
rectangle=>Line
translateX=>Line
translateY=>Line

Constraint Name: rectangle
Choose shape to draw: Line

Current Working Directory: D:\
File to transform: D:\input.pl

Figure 5 The user chooses to edit details about one of the constraints used in the input file.

and 50 correspond to the starting coordinates, the width and the height of the drawn rectangle respectively. Therefore, when translating the rectangle, only one of the starting coordinates needs to be changed.
Whenever a query is entered, the tracer shows the initial objects at the beginning. Through mouse clicks, the user is able to view actions performed in a step-by-step approach. At each step the new objects and the executed rule's name are shown. Figure 6b shows the rectangle after applying the `translateX` rule. The initial rectangle was removed since `translateX` is a simplification rule. Figure 6c visualizes the final rectangle after applying the rule `translateY` on the resulting rectangle. Similarly, the rectangle resulting from applying the `translateX` was removed since `translateY` is a simplification rule.



(a) The initial constraint. **(b)** After applying the rule translateX. **(c)** After applying the rule translateY.

Figure 6 Visualizing the constraints in a step-by-step approach.

6 Conclusions and Future Work

The paper introduced two source-to-source transformers for CHR. Although based on some central ideas introduced before, the new transformers introduce some new techniques of how the transformation could be done. It also overcomes many of the issues faced before. More specifically, the transformer is incorporated within a Java application that parses the input file so that users do not have to worry about translating the program from or into any needed format. Since the transformer and the produced CHR programs were built using the current compiler and CHR runtime system, no modification is needed to be able to use them.

The transformers help visualize the execution of the different enclosed CHR rules in addition to visualizing some of the CHR constraints. Such visual tracers could be useful for many purposes including educational needs and debugging.

It was noticed that both transformers are similar. The technique used is the same in both of them. The only difference is the type of the visual tracer and its corresponding functionalities that are added to the output program. Consequently, for future work, the ultimate goal is to develop a general workbench/engine that facilitates prototyping of source-to-source transformation.

References

1 Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *In Proceedings of 11th Workshop on Logic Programming Environments, 1th*, 2001.

2 Gaudenz Alder. *Design and Implementation of the JGraph Swing Component*, 1.0.6 edition, February 2003. Available at: http://jgraph.sourceforge.net/doc/paper/.

3 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, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.

4 Thom W. 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.

5 Thom W. Frühwirth. Specialization of concurrent guarded multi-set transformation rules. In Sandro Etalle, editor, *LOPSTR*, volume 3573 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2004.

6 Thom W. Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 386–397, 2003.

7 Thom W. Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic constraint handling rules. *Electr. Notes Theor. Comput. Sci.*, 76:115–130, 2002.

8 Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In Michael Leuschel and Andreas Podelski, editors, *PPDP*, pages 25–36. ACM, 2007.

9 Matthias Schmauss. An implementation of CHR in Java. An implementation of CHR in Java, Master Thesis, Institute of Computer Science, LMU, Munich, Germany, November 1999.

10 Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL: A bidirectional Prolog/Java interface. `http://www.swi-prolog.org/packages/jpl/`.

11 Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. Technical Report CW 481, Leuven, Belgium, March 2007.

12 Paolo Tacchella, Maurizio Gabbrielli, and Maria Chiara Meo. Unfolding in chr. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 179–186, New York, NY, USA, 2007. ACM.

13 Peter Van Weert, Jon Sneyers, and Bart Demoen. Aggregates for chr through program transformation. In Andy King, editor, *LOPSTR*, volume 4915 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2007.