CHR-IDE: An Eclipse Plug-In for Constraint Handling Rules

Matthias Rau, Amira Zaki, and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
{matthias-1.rau,amira.zaki,thom.fruehwirth}@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a rule-based programming language which extends a host language such as Prolog, Java or Haskell. Programmers usually write code using generic editors or integrated development environments (IDEs) specific for the host language used. This work presents an IDE developed for the K.U.Leuven CHR(Prolog) system as an Eclipse plug-in. It features a combined editor, executor and visual tracer aimed to enrich a developer's experience.

Keywords: Constraint Handling Rules, Eclipse Plug-in, Editor, Tracer, Integrated Development Environment

1 Introduction

Constraint Handling Rules (CHR) is a committed-choice rule-based programming language based on a set of multi-headed multi-set rewrite rules [11]. CHR is a language extension which uses its host language for the evaluation of built-in constraints. The most common implementation, is that with Prolog, although implementations with Haskell, Java and C also exist. For a developer writing CHR code, it is common to use a generic editor and execute the code externally. Another option could be the use of an integrated development environment (IDE) associated with the host language, however such environments are not customized for CHR. They do not provide special handling for the CHR syntax, neither do they offer a proper CHR tracing mechanism showing interactions with the constraint store.

The K.U.Leuven CHR System is a state-of-the-art CHR system [13]. Their recommended CHR(Prolog) system is that of SWI-Prolog [14]. In this work, the aim is to provide an open-source platform-independent tool for CHR with SWI-Prolog. The idea is realized by creating an integrated development environment for CHR, which offers an editor functionality, an executor and a visual tracer. The main advantage of the tool is integrating these three functionalities together in one place, for the convenience of the developer. The tool also features a visualization of the rule applications as interactions with the constraint store. Constraints inserted and removed are clearly depicted visually in a step-by-step manner. Furthermore, the tool is a developer-assistant, offering syntax highlighting and auto-completion while typing the code. The tool will be of particular interest for people learning programming with CHR as it enhances the user experience during writing, executing and tracing CHR.

In this work, an IDE is created for CHR with SWI-Prolog as a plug-in for Eclipse and is presented as a system demo in this paper. The paper follows by presenting the related work in Section 2, then the features of the plug-in developed are listed in Section 3. The paper concludes in Section 4.

2 Related Work

2.1 Plug-In Development

Eclipse is an open source platform that has become highly popular nowadays as an integrated development environment for Java [2]. It allows flexible program development and integration through its plug-in technology. The development of custom plug-ins which support application-specific functionalities is possible through the plug-in construction toolkit. Several applications have used this toolkit to implement editors for various programming languages [6,8,9,10].

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) is a high-level, committed-choice, rule-based programming language [11,12]. It consists of guarded rules that perform conditional transformation of multi-sets of constraints, known as a constraint store, until a fixed point is reached. CHR is a language extension and depends on a host language for the implementation of built-in constraints; current implementations extend Prolog, Java and Haskell. In this work, CHR is used under SWI-Prolog [14]. CHR utilizes the built-in constraints together with other user-defined CHR constraints. A generalized CHR simpagation rule is given as:

$$\texttt{[}\textit{simpagation-id}\ \texttt{@]}\ H_k \setminus H_r \Leftrightarrow \texttt{[}G\ \texttt{|]}\ B.$$

where H_k and H_r are a conjunction of one or more CHR constraints that are kept/removed respectively. The rule consists of an optional conjunction of host language constraints known as the guard and given by G. Following the partitioning `|` is the body of the rule (B), and it consists of built-in constraints and user-defined CHR constraints. Every rule has an optional unique identifier preceding it, followed by an `@`.

Two other types of rules exist which are special cases of the generalized simpagation rule, namely simplification and propagation rules. A simplification rule does not keep any head constraints, while a propagation rule does not remove any head constraints. These rules are of the form:

$$\texttt{[}\textit{simplification-id}\ \texttt{@]}\ H_r \Leftrightarrow \texttt{[}G\ \texttt{|]}\ B.$$
$$\texttt{[}\textit{propagation-id}\ \texttt{@]}\ H_k \Rightarrow \texttt{[}G\ \texttt{|]}\ B.$$

A complete CHR program consists of the CHR rules, in addition to definitions for some user-defined built-in constraints and other built-in constructs. The complete valid syntax of CHR supported by SWI-Prolog and used for this work can be found in [14].

2.3 Development Tools

Currently, several libraries exist for CHR as an extension of Prolog, Haskell, C and Java. Similarly various development tools exist for these extensions, however no tool to our knowledge features both an integrated editor and a visual tracer customized for CHR.

For SWI-Prolog, Gerhard Röhner [1] has developed an integrated Prolog editor in MS-Windows following the conventions of this platform. It is an editor mainly for Prolog, yet it also supports CHR programs. However the editor is not built with specific handling for CHR, furthermore the tracer in only meaningful for Prolog programs and shows a rather useless trace with CHR.

Prolog Development Tools (ProDT) [7] is a Prolog integrated development environment aiming to be as rich in functionality as java's Eclipse IDE. It provides the developer a single environment to control the development of a Prolog project starting from code edition, to test execution and debugging. This project stands on top of Eclipse's projects to take advantage of its already existent features and its extensibility. The work implemented in this paper is quite similar to ProDT, however we exceed the functionality of ProDT by customizing it to the syntax and use of CHR. Furthermore the debugger offered by ProDT is specialized for Prolog programs, and hence does not work well with CHR. It does not show a constraint store, nor visualize the interactions with it.

Another similar work, was performed for JCHR (CHR extending Java) [3]. An IDE was also developed as an Eclipse plug-in for JCHR of the K.U.Leuven system. This IDE provided a editor interface to facilitate the writing of JCHR programs. Moreover it incorporated a compiler capable of compiling the program written and presenting the output in an external window.

An earlier work proposed a Java library known as JACK (JAva Constraint Kit) [4]. This introduced the use of CHR in Java to write application specific constraint solvers, and provided an interactive tool to visualize the computations. The visualization offered by the tool shows the constraint store as a box, and handles constraints as sub-boxes which are inserted and removed from the box store as the program proceeds. This visualization is similar to the tracing mechanism implemented in our work, however it was implemented on the compiler level.

Another visualization was performed for CHR programs, in order to visualize the execution of the rules [5] through a source-to-source transformation of the CHR programs. Graphical objects are associated with the constraints and the effect of the rule applications are shown on the objects.

3 CHR-IDE

The Eclipse plug-in developed is available online, under `http://pmx.informatik.uni-ulm.de/chr/plugin.zip`. It features three main features which are discussed and illustrated in this section.

3.1 Editor

The aim of the editor is to enhance the user experience whilst programming in CHR. This begins with the addition of a file wizard. The wizard automatically creates a CHR file with the necessary headers. The wizard also allows the programmer to define the user-defined constraints that will be declared with their respective arity. The declaration of these constraints is then automatically written to the created file.

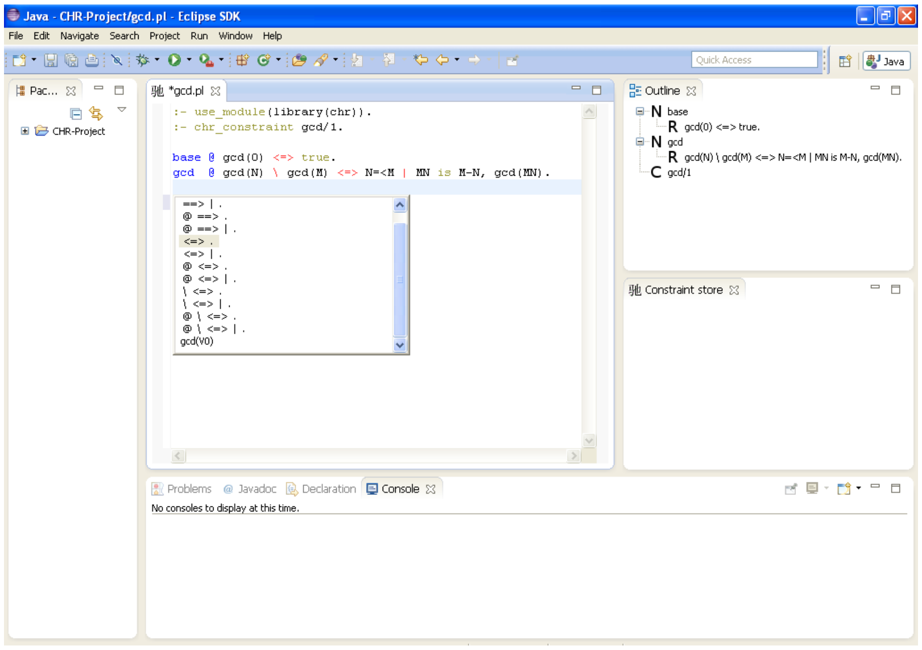
The editor has been enabled with a content assistant functionality. This makes the task of writing CHR rules, much easier. When starting to write any line, it is possible to insert a template for any type of rule. This provides a choice of the three rule types (with and without guards) by inserting the delimiters of the chosen rule type. A snapshot of the editor whilst writing a CHR program is shown in Figure 1.

Additionally an auto-completion feature has been implemented. While typing, one can easily access the list of constraints declared and have them automatically inserted. Auto-completion makes the task of writing CHR rules easier and simpler.

Syntax checking has been enabled of the CHR constructs as listed in the SWI-Prolog manual. The plug-in developed performs a verification of the types code against the CHR grammar. Syntax errors are identified by pattern matching and marked by underlining them. Additionally some warnings are identified, such as assigning a rule identifier to more than one rule.

Furthermore, the editor has been enhanced with a highlighting of the source code written particularly for CHR constructs. The highlighting is done by a rule-based scanner designed precisely for the syntax of CHR. It distinguishes between comments (single-lined and multi-lined ones), rule identifiers, rule delimiters, guard expressions and other code lines which do not contain CHR rules. The syntax highlighting implemented can be seen in Figure 1. The colors chosen can be modified from the preferences of the editor.

An outline of the program is also created continuously as the program evolves and is presented in a side panel. It can be seen in Figure 1 in the top-right corner. Through this outline, it is possible to see the constraints declared with their arity, the rules declared so far and the user-defined built-in predicates defined. Each of these is differentiated from the others by a C, R and P icon respectively (also named rules are marked with a N). This enables a programmer at a glance, to have an overview of the written program. It is useful for large programs which contain mixed ordering of constraint decelerations and rules, by providing a summary of the program constructs.



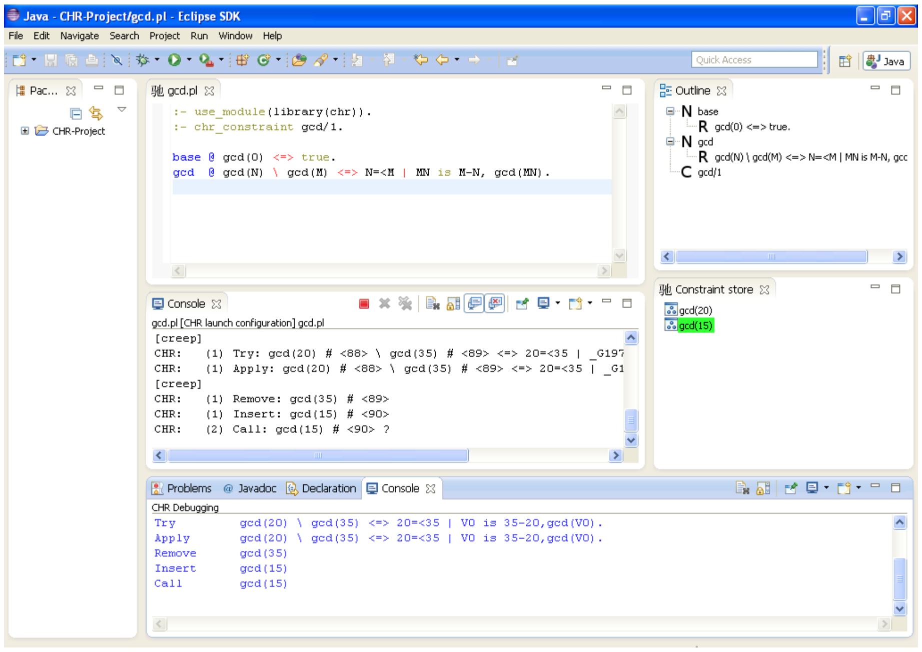
```
:- use_module(library(chr)).
:- chr_constraint gcd/1.

base @ gcd(0) <=> true.
gcd  @ gcd(N) \ gcd(M) <=> N=<M | MN is M-N, gcd(MN).
```

```
==> | .
@ ==> .
@ ==> | .
<=> .
<=> | .
@ <=> .
@ <=> | .
\ <=> .
\ <=> | .
@ \ <=> .
@ \ <=> | .
gcd(V0)
```

```
N base
  R gcd(0) <=> true.
N gcd
  R gcd(N) \ gcd(M) <=> N=<M | MN is M-N, gcd(MN).
C gcd/1
```

No consoles to display at this time.

Fig. 1. Snapshot of CHR-IDE during writing the greatest common divisor in CHR.



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

base @ gcd(0) <=> true.
gcd  @ gcd(N) \ gcd(M) <=> N=<M | MN is M-N, gcd(MN).
```

```
N base
  R gcd(0) <=> true.
N gcd
  R gcd(N) \ gcd(M) <=> N=<M | MN is M-N, gcc
C gcd/1
```

gcd.pl [CHR launch configuration] gcd.pl

```
[creep]
CHR:   (1) Try: gcd(20) # <88> \ gcd(35) # <89> <=> 20=<35 | _G197
CHR:   (1) Apply: gcd(20) # <88> \ gcd(35) # <89> <=> 20=<35 | _G1
[creep]
CHR:   (1) Remove: gcd(35) # <89>
CHR:   (1) Insert: gcd(15) # <90>
CHR:   (2) Call: gcd(15) # <90> ?
```

Constraint store

```
gcd(20)
gcd(15)
```

CHR Debugging

```
Try        gcd(20) \ gcd(35) <=> 20=<35 | V0 is 35-20,gcd(V0).
Apply      gcd(20) \ gcd(35) <=> 20=<35 | V0 is 35-20,gcd(V0).
Remove     gcd(35)
Insert     gcd(15)
Call       gcd(15)
```

Fig. 2. Snapshot of CHR-IDE in debugging/tracer mode, showing constraints currently in the store and a shorter more readable textual trace.

3.2 Executor

The plug-in features a link to SWI-Prolog as the Prolog system installed on the machine. This includes CHR as an extension based on the K.U.Leuven CHR system [13]. Through a console window, all interactions with the SWI-Prolog console are facilitated. Hence a programmer can "run" the code; i.e. consult it. Then the console acts as an interpreter, executing input queries given. The console is identical to that of SWI-Prolog, meaning that backtracking and other options are also enabled.

3.3 Tracer

Another valuable feature of the plug-in developed, is the implementation of a visual code tracer. It is possible using the SWI-library of CHR, to switch on a tracer of the CHR code, by running the program in a debugging mode. This enables a step by step trace of the interactions with the constraint store. The tracer shows how constraints from the query are inserted into the store, in a left to right order. The constraint store is visible in a panel on the right as shown in Figure 1. Then the tracer highlights rule applications whilst writing which constraints are removed and added due to the firing of the rule (as in Figure 2).

The tracer implemented in the plug-in works by parsing the textual CHR trace obtained from SWI (obtained by `chr_trace`). The lines produced of the trace are always indexed with the word `CHR:` followed by keywords which indicate the action that is performed, these are `Insert, Call, Apply, Remove`. The tracer implemented handles line by line the textual trace obtained, and reflects it visually to a constraint store panel. The panel represents the constraint store during the execution of a query. Whenever a constraint is added to the store, it is inserted in the panel as well. Then whenever a rule application takes place, the matching head constraints are highlighted in a different color. Then the rule application occurs, and the result of removing and adding constraints is reflected to the store. This visualization is quite simplistic, however much more readable and understandable than following the textual trace.

In the textual trace, SWI automatically renames local variables with an internal naming convention (an underscore, a letter then followed by a sequence of numbers). During a trace of multiple variables, it becomes quite tedious to follow the variables. Thus to enhance the readability of the variables in the constraint store, the tracer implemented renames variables to short alphabetical names.

Furthermore to extend the readability, a supplementary shorter and more readable textual trace is displayed in another console (as seen in the bottom on Figure 2). This includes a textual representation of the trace, with the modified variable names and having redundant constructs removed. With experience, it has been seen that this shorter trace is more understandable than the original longer one.

4 Conclusion

In this work, a complete integrated development environment (IDE) was implemented for the K.U.Leuven CHR system offered with SWI-Prolog. The IDE was created as a plug-in for Eclipse, which offers a combined editor, executor and visual tracer functionality. The editor provides customized syntax highlighting, syntax error checking, auto-completion and a content assistant functionality. These features enrich the developer experience whilst writing CHR. The plug-in integrates a means to compile and execute the written programs. It uses the SWI-Prolog installation on the machine. The plug-in also features a tracer, capable of visualizing the execution of the rules by graphically displaying the constraint store and coloring the interactions with it.

The plug-in is still under development and several extensions will be implemented to increase its usefulness. One such feature would be to integrate the compiler detected errors and warnings. Compiler detected syntax errors could be reflected back in the IDE by highlighting the error line, moreover fixes can be suggested for warnings such as removing rules that never fire. Another next step, would be handling type and mode declarations, and integrating some static type checking in the IDE and proposing type decelerations heuristically based on the rules. The tracer could also be enhanced by allowing breakpoints on rules or occurrences of constraints. Furthermore, we aim to integrate the visualization of [5] to depict the constraints and their interactions visually by graphical components through a source-to-source transformation of the code.

References

1. SWI-Prolog-Editor for Windows. http://lakk.bildung.hessen.de/netzwerk/faecher/informatik/swiprolog/indexe.html. Accessed: 24.05.2013.
2. The Eclipse Foundation open source community wesbite. http://www.eclipse.org/. Accessed: 24.05.2013.
3. Slim Abdennadher and Shehab Fawzy. JCHRIDE: An Integrated Development Environment for JCHR. In Sibylle Schwarz, editor, *WLP '08: Proc. 22nd Workshop on (Constraint) Logic Programming*, pages 1–6, Dresden, Germany, September 2008.
4. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauß. JACK: A Java Constraint Kit. In *WFLP '01: Proceedings of 10th International Workshop on Functional and (Constraint) Logic Programming*, volume 64, pages 1–17, 2002.
5. Slim Abdennadher and Nada Sharaf. Visualization of CHR through Source-to-Source Transformation. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 109–118. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.
6. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Ranjit Majumdar. An eclipse plug-in for model checking. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 251–255, 2004.
7. Claudio Cancinos and Serrano Carolina. Prolog Development Tools (ProDT). http://prodevtools.sourceforge.net/index.html. Accessed: 24.05.2013.

8. Gerardo Canfora and Luigi Cerulo. Jimpa: An eclipse plug-in for impact analysis. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '06, pages 341–342, 2006.
9. Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. Jasper: an eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 65–69, 2006.
10. Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, 2005.
11. Thom Frühwirth. *Constraint handling rules.* Cambridge University Press, 2009.
12. Thom Frühwirth and Frank Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis.* Books on Demand, March 2011.
13. Tom Schrijvers and Bart Demoen. The k.u.leuven chr system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
14. Jan Wielemaker, Thom Frühwirth, Leslie De Koninck, Markus Triska, and Marcus Uneson. *SWI Prolog Reference Manual 6.2.2.* Books on Demand, September 2012.