

**Proceedings of the
7th International Workshop
on Constraint Handling Rules**

Peter Van Weert and Leslie De Koninck

Report CW 588, May 2010



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

**Proceedings of the
7th International Workshop
on Constraint Handling Rules**

Peter Van Weert and Leslie De Koninck

Report CW 588, May 2010

Department of Computer Science, K.U.Leuven

Abstract

This volume contains the papers presented at CHR 2010, the Seventh International Workshop on Constraint Handling Rules held at the occasion of ICLP 2010, part of FLoC 2010, on 20 July 2010 in Edinburgh, Scotland.

Preface

This volume contains the papers presented at CHR 2010, the Seventh International Workshop on Constraint Handling Rules. The workshop was held on 20 July 2010 in Edinburgh, Scotland at the occasion of the International Conference on Logic Programming (ICLP 2010), part of the Federated Logic Conference (FLoC 2010).

The Constraint Handling Rules (CHR) language has become a major declarative specification and implementation language for constraint reasoning algorithms and applications. Algorithms are often specified using inference rules, rewrite rules, proof rules, or logical axioms that can be directly written in CHR. Based on first-order predicate logic, this clean semantics of CHR facilitates non-trivial program analysis and transformation. For more information on the language, we refer the interested reader to the CHR website: <http://dtai.cs.kuleuven.be/CHR/>.

Previous Workshops on Constraint Handling Rules were organized in Ulm, Germany (2004), Sitges, Spain (2005) at ICLP, Venice, Italy (2006) at ICALP, Porto, Portugal (2007) at ICLP, Hagenberg, Austria (2008) at RTA, and Pasadena, CA, USA (2009) at ICLP. All proceedings are freely available from <http://dtai.cs.kuleuven.be/CHR/workshop.shtml>. The CHR workshop aims to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments.

This year, there were 6 submissions, each of which was reviewed by 3 program committee members. The committee decided to accept all papers. Next to the accepted papers, the program featured two invited talks. Mark Proctor, lead of the JBoss Drools project, gave a tutorial on the JBoss Drools Business Logic integration Platform, and Matt Lilley of the New Zealand company SecuriteEase on using Prolog and CHR in a commercial setting. Also included in the program were a tutorial on CHR by Thom Frühwirth, an introduction on CHRiSM by Jon Sneyers, and an additional demonstration of a CHR-based application that offers long-term routing for autonomous sailing boats.

We thank the authors of the submitted papers, the invited speakers and program committee members, as well as the organizers of FLoC 2010 and ICLP 2010 for making this workshop possible. We would also like to thank Phil Scott, the FLoC 2010 workshop coordinator, and Andrei Voronkov of EasyChair for their assistance on the practical side of organizing the workshop.

May 2010

Leslie De Koninck
Peter Van Weert

Workshop Organization

Program Chairs

Leslie De Koninck, NICTA and University of Melbourne, Australia
Peter Van Weert, K.U.Leuven, Belgium

Program Committee

Sebastian Brand, NICTA and University of Melbourne, Australia
Henning Christiansen, Roskilde University, Denmark
Veronica Dahl, Simon Fraser University, Canada
Thom Frühwirth, Ulm University, Germany
Marco Gavanelli, University of Ferrara, Italy
Rémy Haemmerlé, Universidad Politécnica de Madrid, Spain
Maria Chiara Meo, “Gabriele d’Annunzio” University, Italy
Paolo Pilozzi, K.U.Leuven, Belgium
Frank Raiser, Ulm University, Germany
Jairson Vitorino, Federal University of Pernambuco, Brazil
Armin Wolf, Fraunhofer FIRST, Germany

Table of Contents

A State Equivalence and Confluence Checker for CHR	1
<i>Johannes Langbein, Frank Raiser, and Thom Frühwirth</i>	
Join Ordering for Constraint Handling Rules: Putting Theory into Practice	9
<i>Peter Van Weert</i>	
The Viterbi Algorithm expressed in Constraint Handling Rules . .	17
<i>Henning Christiansen, Christian Theil Have, Ole Torp Lassen, and Matthieu Petit</i>	
Result-directed CHR Execution	25
<i>Jon Sneyers</i>	
Generic and Extensible Automatic Test Data Generation for Safety Critical Software with CHR	33
<i>Ralf Gerlich</i>	
MTSeq: Multi-touch-enabled CHR-based Music Generation and Manipulation	41
<i>Florian Geiselhart, Frank Raiser, Jon Sneyers, and Thom Frühwirth</i>	

A State Equivalence and Confluence Checker for CHR

Johannes Langbein, Frank Raiser, and Thom Frühwirth

Faculty of Engineering and Computer Science, Ulm University, Germany
`firstname.lastname@uni-ulm.de`

Abstract. Analyzing confluence of CHR programs manually can be an impractical and time consuming task. Based on a new theorem for state equivalence, this work presents the first tool for testing equivalence of CHR states. As state equivalence is an essential component of confluence analysis, we apply this tool in the development of a confluence checker that overcomes limitations of existing checkers. We further provide evaluation results for both tools and detail their modular design, which allows for extensions and reuse in future implementations of CHR tools.

1 Introduction

Constraint Handling Rules (CHR) [1] is a declarative, multiset- and rule-based programming language suitable for powerful program analysis.

One such property a CHR program can be analyzed for is called *confluence*. For a confluent program, it is guaranteed that we always get the same result for a given query, independently of the order of rule applications and the order of constraints in the query. Furthermore, confluent programs are parallelizable without changing their source code [1].

There is a decidable, sufficient, and necessary criterion for confluence of terminating CHR programs [2]. It is based on the joinability of so-called *critical pairs*. However, proving confluence of a larger program manually can quickly become infeasible as there is a combinatorial explosion in the number of critical pairs with program size. Hence, confluence checking is preferably done using a software tool.

Equivalence of CHR states is a fundamental notion used in the criterion for confluence. Recent work [3, 4] has led to an axiomatic definition of state equivalence alongside a decidable and sufficient criterion.

Based on this work, we implemented a checker for equivalence of CHR states. Furthermore, we used this checker to implement a new tool for confluence checking, which overcomes limitations (cf. Section 4.2) of existing confluence checkers [2, 5]. The checkers for state equivalence and confluence together with some example CHR programs and test-cases are available under GNU GPL and can be downloaded from [6]. They can be used for any terminating CHR program which does not contain propagation rules. Furthermore, only the built-ins `true`, `false`, and `=/2` are supported (cf. Section 5.1).

We make the following contributions:

- We present the first implementation of a test for CHR state equivalence (Section 3.1).
- Based on this, we describe our implementation of a new confluence checker (Section 3.2).
- We evaluate our confluence checker with the union-find implementations presented in [5] and compare it to previous implementations (Section 4).
- Finally, we point out possible future extensions of our checkers (Section 5.1).

2 Preliminaries

Constraint Handling Rules distinguishes two kinds of constraints: *CHR constraints* and *built-in constraints*. We assume reasoning on built-in constraints to be possible through a satisfaction-complete and decidable constraint theory \mathcal{CT} . We use the following definitions of CHR states and state equivalence from [3].

Definition 1 (CHR States). *A CHR state σ is a tuple $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$. The goal \mathbb{G} is a multiset of CHR constraints. The built-in constraint store \mathbb{B} is a conjunction of built-in constraints. \mathbb{V} is a set of global variables. A variable $v \in (\mathbb{B} \cup \mathbb{G})$ is called a local variable iff $v \notin \mathbb{V}$. A variable $v \in \mathbb{B}$ is called a strictly local variable iff $v \notin (\mathbb{V} \cup \mathbb{G})$. We use Σ to denote the set of all states.*

Definition 2 (State Equivalence). *Equivalence between CHR states is the smallest equivalence relation \equiv over CHR states that satisfies the following conditions:*

1. (Equality as Substitution)
 $\langle \mathbb{G}; X \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G} [X/t]; X \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle$
2. (Transformation of the Constraint store)
If $\mathcal{CT} \models \forall (\exists \bar{s}. \mathbb{B} \leftrightarrow \exists \bar{s}'. \mathbb{B}')$ where \bar{s}, \bar{s}' are the strictly local variables of \mathbb{B}, \mathbb{B}' , respectively, then $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}'; \mathbb{V} \rangle$
3. (Omission of Non-Occuring Global Variables)
If X is a variable that does not occur in \mathbb{G} or \mathbb{B} then $\langle \mathbb{G}; \mathbb{B}; \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$
4. (Equivalence of failed states)
 $\langle \mathbb{G}; \perp; \mathbb{V} \rangle \equiv \langle \mathbb{G}'; \perp; \mathbb{V} \rangle$

The following sufficient and decidable criterion for state equivalence was introduced in [3].

Theorem 1 (Criterion for \equiv). *Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ and $\sigma' = \langle \mathbb{G}'; \mathbb{B}'; \mathbb{V} \rangle$ be CHR states with local variables \bar{y} and \bar{y}' that have been renamed apart. Then*

$$\sigma \equiv \sigma' \text{ iff } \mathcal{CT} \models \forall (\mathbb{B} \rightarrow \exists \bar{y}'. ((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall (\mathbb{B}' \rightarrow \exists \bar{y}. ((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

For $\mathbb{G} = \{c_1, \dots, c_n\}$ and $\mathbb{G}' = \{c'_1, \dots, c'_n\}$ the expression $(\mathbb{G} = \mathbb{G}')$ denotes the following set of equations: $c_{\tau(1)} = c'_1 \wedge \dots \wedge c_{\tau(n)} = c'_n$ for a permutation τ .

A CHR program \mathcal{P} is a set of rules of the following form.

Definition 3 (CHR Rule). *A CHR (simpagation) rule is of the form*

$$r @ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b$$

where H_1 and H_2 are multisets of user-defined constraints, called the kept head and removed head, respectively. The guard G is a conjunction of built-in constraints and the body consists of a conjunction of built-in constraints B_b and a multiset of user-defined constraints B_c . The rule name r is optional and may be omitted along with the $@$ symbol. If H_1 is empty, we call the rule a simplification rule.

In this paper, we use the following equivalence-based operational semantics ω_e which was established in [3].

Definition 4 (ω_e Transitions). *For a CHR program \mathcal{P} , the state transition system $(\Sigma/\equiv, \mapsto)$ is defined as follows. The transition is based on a variant of a rule r in \mathcal{P} such that its local variables are disjoint from the variables occurring in the pre-transition state.*

$$\frac{r @ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b}{\langle H_1 \uplus H_2 \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle \mapsto^r \langle H_1 \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle}$$

When the rule r is clear from the context or not important, we may write \mapsto rather than \mapsto^r . By \mapsto^* , we denote the reflexive-transitive closure of \mapsto . For two states σ and σ' we may also write $\sigma \mapsto \sigma'$ instead of $[\sigma] \mapsto [\sigma']$.

Confluence is a property which guarantees that all possible computations for a given goal result in equivalent final states. For confluent programs, the order of constraints in a goal and the order of rules in the program does not matter [2]. We define confluence using the following definition of joinability.

Definition 5 (Joinability). *Two CHR states σ_1 and σ_2 are called joinable if there exist states σ'_1 and σ'_2 such that $\sigma_1 \mapsto^* \sigma'_1$ and $\sigma_2 \mapsto^* \sigma'_2$ with $\sigma'_1 \equiv \sigma'_2$*

Definition 6 (Confluence). *A CHR program is confluent if for all states σ , σ_1 , σ_2 :*

$$\text{If } \sigma \mapsto^* \sigma_1 \text{ and } \sigma \mapsto^* \sigma_2, \text{ then } \sigma_1 \text{ and } \sigma_2 \text{ are joinable.}$$

We cannot check joinability starting from all possible states σ , as in general there are infinitely many such states. However, there exists a decidable, sufficient and necessary criterion for confluence of terminating CHR programs [2], which is based on joinability of so-called critical pairs. The criterion states, that for terminating programs, we can restrict the joinability test to those critical pairs, of which only a finite number exists. For critical pairs, we use the following, adapted definition from [7]:

Definition 7 (Critical Pair). *Given two (not necessarily different) rules $r_1 @ H_{11} \setminus H_{21} \Leftrightarrow G_1 \mid B_{c1} \uplus B_{b1}$ and $r_2 @ H_{12} \setminus H_{22} \Leftrightarrow G_2 \mid B_{c2} \uplus B_{b2}$ whose variables have been renamed apart.*

The tuple (σ_1, σ_2) with

$$\begin{aligned} \sigma_1 &= \langle B_{c1} \uplus ((H_{r1}^\Delta \uplus H_{r2}^\Delta \uplus H_{r1}^\cap) - H_{21}); (H_{r1}^\cap = H_{r2}^\cap) \wedge B_{b1} \wedge G_1 \wedge G_2; \mathbb{V} \rangle \\ \sigma_2 &= \langle B_{c2} \uplus ((H_{r1}^\Delta \uplus H_{r2}^\Delta \uplus H_{r1}^\cap) - H_{22}); (H_{r1}^\cap = H_{r2}^\cap) \wedge B_{b2} \wedge G_1 \wedge G_2; \mathbb{V} \rangle \end{aligned}$$

where

$$\begin{aligned} H_{r1}^\Delta \uplus H_{r1}^\cap &= H_{11} \uplus H_{21} \\ H_{r2}^\Delta \uplus H_{r2}^\cap &= H_{12} \uplus H_{22} \\ \mathbb{V} &= \text{vars}(H_{r1}^\Delta \wedge H_{r2}^\Delta \wedge H_{r1}^\cap \wedge G_1 \wedge G_2) \end{aligned}$$

is called a critical pair of r_1 and r_2 if $H_{r1}^\cap \neq \emptyset$ and $\mathcal{CT} \models (H_{r1}^\cap = H_{r2}^\cap) \wedge G_1 \wedge G_2$.

Here, the equality $=$ between multisets of head constraints has the same meaning as in Theorem 1.

It suffices to show joinability of critical pairs to show confluence of a terminating program.

Theorem 2 (Criterion for confluence [2]). *A terminating CHR program \mathcal{P} is confluent iff all critical pairs of all rules of \mathcal{P} are joinable.*

In [2] this theorem has been proven for CHR programs consisting only of simplification rules. However, as we do not consider propagation rules and as simplification rules are only an abbreviation for simplification rules, the theorem remains valid. Furthermore, this allows us to ignore the token-store from [8] and thus apply the theorem using CHR states according to Definition 1.

Confluence can be tested by posing the two states σ_1 and σ_2 of a critical pair as query to the CHR program, as the following argumentation shows: If the two queries

result in two states σ'_1 and σ'_2 with $\sigma'_1 \equiv \sigma'_2$, we have shown joinability of the critical pair. If the two queries result in two non-equivalent states σ'_1 and σ'_2 , there should be transitions $\sigma_1 \mapsto^* \sigma''_1$ and $\sigma_2 \mapsto^* \sigma''_2$ such that $\sigma''_1 \equiv \sigma''_2$, if the program was confluent as the order of rule application should not matter in this case. However, if the computation stops with σ'_1 and σ'_2 as final states, no more rules are applicable, in particular there are no transitions leading to equivalent states σ''_1 and σ''_2 . This means we have found a counterexample for confluence of the program.

3 State Equivalence and Confluence Checkers

The checkers for state equivalence and confluence are implemented in SWI-Prolog. They are organized in two different modules named `stateequiv` and `conflcheck` which define the predicates for state equivalence and confluence, respectively.

3.1 State Equivalence

The module `stateequiv` is an implementation of Theorem 1 for the equivalence relation \equiv over CHR states. In this module, CHR states (cf. Definition 1) are represented by Prolog terms of the form `state(G, B, V)`, where `G`, `B`, and `V` are lists representing the goal store, the built-in store and the global variables of the state. CHR constraints are represented by Prolog terms, built-in constraints by the terms `=/2`, `true`, and `false` (which are the only supported built-ins, see Section 5.1) while variables are represented as Prolog variables. The empty goal and built-in stores \top as well as the empty set of variables are represented as empty Prolog lists.

Example 1. The CHR state $\langle \{c(X)\}, X = 1, \{X\} \rangle$ is represented by the term `state([c(X)], [X=1], [X])`.

The predicate `equivalent_states/2` takes two `state` terms representing CHR states σ and σ' as arguments and succeeds if and only if $\sigma \equiv \sigma'$.

Example 2. The goal

```
equivalent_states(state([c(X)], [X=1], [X]), state([c(1)], [X=1], [X]))
```

succeeds while the following goal fails:

```
equivalent_states(state([c(X)], [], [X]), state([c(X)], [X=1], [X]))
```

3.2 Confluence

Based on the module for state equivalence, the module `conflcheck` implements the criterion for confluence from Theorem 2. The confluence checker creates all possible critical pairs of all rules in the program according to Definition 7. It checks them for joinability by posing the two states separately as query to the CHR program and retrieving the two resulting states, which are tested for equivalence by `equivalent_states/2`.

The entire list of critical pairs is created before each individual critical pair is tested for joinability. This allows for filtering out critical pairs which are variants or symmetrical to other critical pairs. The list is also filtered to remove critical pairs whose states are already equivalent. Only the remaining critical pairs are tested for joinability.

The module `conflcheck` defines the predicate `check_confluence/1` which takes the path to a CHR program file as argument and checks this program for confluence. The predicate always succeeds, printing either a message of success or a list of non-joinable critical pairs.

Example 3. Consider the following program simulating destructive assignment, which can also be found in `examples/mem.pl` in the package available at [6]:

```
assign(V,N), cell(V,0) <=> cell(V,N).
```

This program is not confluent as it has two non-joinable critical pairs. The call `check_confluence('examples/mem.pl')` outputs information about the two non-joinable critical pairs as follows (shortened):

```
Checking confluence of CHR program in examples/mem.pl...
```

```
The following critical pair is not joinable:
```

```
state([cell(A, B), cell(C, D)], [C=A, D=E], [C, D, F, A, E, B])
state([cell(C, F), cell(A, E)], [C=A, D=E], [C, D, F, A, E, B])
```

```
...
```

```
The following critical pair is not joinable:
```

```
state([assign(A, B), cell(C, D)], [C=A, E=F], [C, D, E, A, B, F])
state([assign(C, D), cell(A, B)], [C=A, E=F], [C, D, E, A, B, F])
```

```
...
```

```
The CHR program in examples/mem.pl is NOT confluent!
```

```
2 non-joinable critical pair(s) found!
```

The predicate `check_confluence/3` works the same way but only considers critical pairs of two (not necessarily different) rules in the program. The module `conflcheck` also offers predicates which return the number of non-joinable critical pairs instead of printing them. Those predicates can be used to call the confluence checker from other programs. Details about the mentioned predicates and additionally ones are given in the manual of the confluence checker [6].

4 Evaluation and Related Work

In this section, we describe the results of applying our confluence checker to programs which have already been analyzed for confluence as well as we compare our implementation to existing confluence checkers for CHR programs.

4.1 Confluence of Union-Find

In addition to unit-tests with classic CHR programs, which are defined in the file `tests.pl` in the package available at [6], we evaluated our confluence checker with the CHR implementations of the union-find algorithm from [5]; on the one hand to test our checkers with programs yielding more critical pairs, on the other hand to compare the results of our confluence checker to a previous confluence analysis from [5].

We were able to confirm the number of critical pairs of the implementations in `ufd_basic.pl`, `ufd_basic1.pl`, and `ufd_rank.pl` for those rules our confluence checker is applicable to.

However, checking the confluence of the parallelized optimal union-find implementation in `ufd_found_compr.pl`, we found differing numbers of non-joinable critical pairs for some rules:

- The rule `findroot1 @ root(A,_) \ find(A,X) <=> found(A,X)` has two non-trivial critical pairs that we both found to be joinable in contrary to one non-joinable critical pair mentioned in [5].
- For the rule `compress @ foundc(C,X) \ A~>B, compr(A,X) <=> A~>C` we found the following four non-joinable critical pairs, in contrary to [5], where only three non-joinable critical pairs were found:

$$\begin{aligned}
& (\text{foundc}(C,X) \wedge A \rightsquigarrow C \wedge A \rightsquigarrow D, \\
& \text{foundc}(C,X) \wedge A \rightsquigarrow C \wedge A \rightsquigarrow B) \\
& (\text{foundc}(C,X) \wedge \text{foundc}(D,E) \wedge A \rightsquigarrow C \wedge \text{compr}(A,E), \\
& \text{foundc}(C,X) \wedge \text{foundc}(D,E) \wedge A \rightsquigarrow D \wedge \text{compr}(A,X)) \\
& (\text{foundc}(C,X) \wedge \text{foundc}(D,X) \wedge A \rightsquigarrow C, \\
& \text{foundc}(C,X) \wedge \text{foundc}(D,X) \wedge A \rightsquigarrow D) \\
& (\text{foundc}(C,X) \wedge A \rightsquigarrow C \wedge \text{foundc}(D,X) \wedge A \rightsquigarrow E, \\
& \text{foundc}(C,X) \wedge A \rightsquigarrow D \wedge \text{foundc}(D,X) \wedge A \rightsquigarrow B)
\end{aligned}$$

- The rule `linkeq1c @ found(A,X), found(A,Y), link(X,Y) <=>`
`foundc(A,X), foundc(A,Y)` leads to 18 non-joinable critical pairs according to [5]. However, there are only 13 different possibilities to overlap the head constraints of this rule leading to six non-joinable critical pairs.

Using our new tools, we were able to correct some of the numbers of non-joinable critical pairs. We suppose the differing numbers to be due to typos or an error in the existing implementation as it can be easily shown that our above-mentioned numbers are correct. However, despite our findings, the general results presented in [5] remain correct in that the programs are not confluent.

4.2 Related Work

To our knowledge there are two existing implementations of confluence analysis for CHR programs. The tool used to check confluence of the union-find implementations in [5] requires the CHR program to be represented as Prolog facts in the source code of the tool. Another existing confluence checker [2], whose sources are available to the authors, is limited to single-headed simplification rules. Furthermore, the existing tools check joinability of critical pairs based on the notion of variants rather than state equivalence. Our confluence checker overcomes those limitations as it supports multi-headed and simpagation rules and directly parses a CHR source file as input. Also, it checks joinability based on state equivalence, according to Definition 5.

Both existing confluence checkers require manual changes in the source code of the analyzed CHR program in order to process built-in constraints. Our implementation does not need any changing of the source file to process built-in constraints. However, as a trade-off, it is restricted to the built-ins `=/2`, `true`, and `false` as of now (cf. Section 5.1).

5 Conclusion and Future Work

We have presented the first implementation of a program for state equivalence testing and a new confluence checker based on an axiomatic definition of state equivalence. We used a modular design for our checkers to allow for extensions and reuse of our code. We have tested them with unit-tests and existing CHR implementations of the union-find algorithm, re-evaluated the previously published results about their numbers of critical pairs, and compared our confluence checker to existing implementations.

5.1 Future Work

Due to our definition of states which does not provide any means to express the propagation history of the operational semantics ω_t [1], our confluence checker does

not work for programs containing propagation rules. Taking critical pairs from propagation rules alongside the propagation history into account is one possible solution for this. Alternatively, the confluence checker can be extended to support persistent constraints [9, 4].

To allow arbitrary Prolog predicates as built-in constraints, the checkers for state equivalence and confluence need to be extended to check logical entailment. The current restriction to the built-ins `=/2`, `true`, and `false` arises from the fact that Prolog in general requires the arguments of built-ins to be ground, while CHR states and especially critical pairs can contain unbound variables.

An extension to the notion of confluence is the so-called *observable confluence* [7]. The criterion for observable confluence only considers critical pairs satisfying an invariant. Our confluence checker can be adapted to this notion of confluence: the predicate `process_cps/2` can be used for any kind of modification of critical pairs and thus can be extended to test and alter each critical pair according to a given invariant. Alternatively, the addition of an interactive mode could enable the user to check and modify each critical pair before it is tested for joinability.

In [10], a criterion for operational equivalence of CHR programs, which also relies on joinability, is given. Our implementations of critical pair generation and state equivalence can be adapted and used to implement a checker for operational equivalence.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Abdennadher, S., Frühwirth, T., Meuss, H.: On Confluence of Constraint Handling Rules. In Freuder, E., ed.: Principles and Practice of Constraint Programming, Second International Conference, CP 96. Volume 1118 of Lecture Notes in Computer Science., Springer-Verlag (1996) 1–15
3. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR States Revisited. In Raiser, F., Sneyers, J., eds.: Sixth International Workshop on Constraint Handling Rules (CHR). (2009) 34–48
4. Betz, H., Raiser, F., Frühwirth, T.: A Complete and Terminating Execution Model for Constraint Handling Rules. In: Logic Programming, 26th International Conference, ICLP 2010. (2010) accepted.
5. Frühwirth, T.: Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In Gabbrielli, M., Gupta, G., eds.: Logic Programming, 21st International Conference, ICLP 2005. Volume 3668 of Lecture Notes in Computer Science., Springer-Verlag (2005) 113–127
6. Langbein, J.: A State Equivalence and Confluence Checker for CHR, Prolog sources. <http://www.uni-ulm.de/en/in/pm/research/topics/chr/info/downloads.html>
7. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable Confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. Volume 4670 of Lecture Notes in Computer Science., Springer-Verlag (2007) 224–239
8. Abdennadher, S.: Operational Semantics and Confluence of Constraint Propagation Rules. In: Principles and Practice of Constraint Programming, Third International Conference, CP97. Volume 1330 of Lecture Notes in Computer Science., Springer-Verlag (1997) 252–266
9. Betz, H., Raiser, F., Frühwirth, T.: Persistent Constraints in Constraint Handling Rules. In: (Constraint) Logic Programming, 23rd Workshop, WLP 2009, Institute of Computer Science, Potsdam University (2009)
10. Abdennadher, S., Frühwirth, T.: Operational Equivalence of CHR Programs and Constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming, Fifth International Conference, CP 99. Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57

Join Ordering for Constraint Handling Rules

Putting Theory into Practice

Peter Van Weert*

Department of Computer Science, K.U.Leuven, Belgium
Peter.VanWeert@cs.kuleuven.be

Abstract. Join ordering is the NP-complete problem of finding the optimal order in which the different conjuncts of multi-headed rules are joined. Join orders are the single most important determinants for the runtime complexity of CHR programs. Nevertheless, all current systems use ad-hoc join ordering heuristics, often using greedy, very error-prone algorithms. As a first step, Leslie De Koninck and Jon Sneyers therefore worked out a more realistic, flexible formal cost model. In this work-in-progress paper, we show how we created a first practical implementation of static join ordering based on their theoretical model.

1 Introduction

Constraint Handling Rules (CHR) [6] is an elegant, very high-level programming language based on multi-headed guarded rules. Originally designed for the declarative specification of constraint solvers, CHR is increasingly used as a general purpose programming language, in a wide range of applications [15]. A considerable amount of research is devoted to the optimizing compilation and execution of CHR programs [5, 11, 19, 22], and efficient, state-of-the-art implementations exist for Prolog [11, 12], HAL [5, 8], Java [21], Haskell, and C [23].

The most critical part of any rule-based system is the search for matching partner constraints to form applicable rule instances, given an active—typically just added—CHR constraint. To prune this search space many techniques are used, including loop-invariant code motion (e.g. testing guards as soon as possible) and constraint store indexing (cf. Example 1). Their applicability and effectiveness is almost always completely determined by the order in which the partner constraints are joined.

Example 1. The following rule occurs in the CHR-based RAM simulator of [14]:

$$\text{pc}(L), \text{mem}(A, X) \setminus \text{prog}(L, \text{ADD}, B, A), \text{mem}(B, Y) \Leftrightarrow \text{mem}(A, X+Y), \text{pc}(L+1).$$

It implements the ADD instruction of the simulated RAM machine. The different CHR constraints model the RAM machine’s program counter (`pc/1`), memory (`mem/2`), and program (`prog/4`).

Suppose a new `pc(L)` constraint is added. To determine whether the above rule is applicable, a naive implementation would match the different conjuncts of the head in textual order. This entails enumerating all `mem/2` constraints, for each of them checking whether a suitable `prog/4` constraint is in the store. Even if e.g. a hash- or array-based index is used to check for matching `prog/4` constraints in $\mathcal{O}(1)$ time, this process remains linear in the size of the RAM machine’s memory.

With the correct join order, the runtime would first look up a matching `prog/4` constraint using the L value known from the active `pc(L)` constraint, and only then retrieve the two `mem/2` constraints. This way, given proper indexing, the evaluation of the rule occurs in optimal constant time, instead of the naive linear time.

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).

Finding optimal join orders is thus quintessential for the optimal time complexity of most CHR programs. The join ordering problem, however, is NP-complete [9], and may moreover depend on dynamic properties such as the size of the constraint store, the selectivity of guards, etc. Current state-of-the-art optimizing compilers therefore use ad-hoc heuristics to determine join order, mostly based on those proposed in [5, 8]. They moreover mostly use ad-hoc algorithms to minimize the estimated cost.

As a first step towards more effective join ordering for CHR, [2, 3, 13] therefore worked out a reasonable, more realistic cost formula, and discussed in detail how to heuristically approximate it either statically or dynamically. They moreover proposed several techniques adapted from database literature [10, 16, 18] for implementing join ordering based on their model. Unfortunately, their descriptions are very sketchy and contain errors, which we will point out and correct in this paper.

This work-in-progress paper thus represents a necessary second step, transforming the theoretical principles of [2, 3, 13] into correct, practical join ordering algorithms. We first explain three join orderers we implemented for JCHR2 (Sections 3–4), an upcoming new CHR system for Java [21]. Next, Section 5 briefly lists some considerations for implementing the more efficient ‘KBZ’ algorithm proposed in [2, 3, 13], and Section 6 compares with related work. The next step, part of future work (Section 8), will involve validating, fine-tuning, and improving the cost formula, our heuristics and our algorithms based on more extensive experimentation.

2 Problem Statement

In this section, we very briefly and informally reconstruct the cost formula derived in [2, 3, 13]. More rigorous definitions can be found in these references.

Slightly simplified¹ and reordered, and under a number of reasonable assumptions (e.g. only $\mathcal{O}(1)$ equality indexes are used, and all remaining—so-called *a posteriori*—guards are also evaluated in constant time) and restrictions (e.g. nested loop joins only), the cost of matching n heads according to a given join order Θ is:

$$C_{\Theta}^{[1..n]} = \sum_{j=1}^n |\mathcal{J}_{\Theta}^{j-1}| \cdot \mu^{\Theta}(j) = \sum_{j=1}^n \prod_{k=1}^{j-1} (\mu^{\Theta}(k) \cdot \sigma_{\star}^{\Theta}(k)) \cdot \mu^{\Theta}(j) \quad (1)$$

with (all defined assuming partners are joined in the order determined by Θ):

- $|\mathcal{J}_{\Theta}^k| = |\mathcal{J}_{\Theta}^{k-1}| \cdot \mu^{\Theta}(k) \cdot \sigma_{\star}^{\Theta}(k)$ the size of a partial join: the number of CHR constraint tuples that match the first k heads; we call these *k-tuples*;
- $\mu^{\Theta}(k)$ the (average) *multiplicity*: the average number of constraints that satisfy the k 'th partner's *a-priori guards*—the guards that are tested a priori using a constraint index—per $(k-1)$ -tuple for which at least one k -tuple exists; and
- $\sigma_{\star}^{\Theta}(k)$ the (average) *selectivity*: the average percentage of these k -tuples that satisfy the k 'th partner's *a-posteriori guards*—the remaining guards.

Our join ordering problem is thus finding a join order Θ that minimizes the cost formula (1). We refer to [2, 3, 13] on detailed discussions on how to (statically) estimate the $\mu^{\Theta}(k)$ and $\sigma_{\star}^{\Theta}(k)$ factors.

3 Exhaustive Algorithms

3.1 Branch and bound join ordering

The most straightforward join ordering algorithm exhaustively enumerates all $n!$ possible join orderings. It can be viewed as traversing a tree with the empty join at the root, and complete join orderings at the leaves, in a depth-first, left-to-right

¹ Concretely, for simplicity, we ignore the $\sigma_{\text{eq}}^{\Theta}(k)$ factor of the actual cost formula. This is reasonable, since any static estimate assumes $\sigma_{\text{eq}}^{\Theta}(k) = 1$ (cf. [2, 3, 13] for details).

order. For this, it is more convenient to rewrite (1) as follows:

$$C_{\Theta}^{[1..n]} = \sum_{j=1}^n \prod_{k=1}^j (\sigma_{\star}^{\Theta}(k-1) \cdot \mu^{\Theta}(k))$$

It then becomes apparent that this sum can be efficiently computed incrementally:

$$\begin{cases} C_{\Theta}^{[1..0]} = 0 \\ C_{\Theta}^{[1..i]} = C_{\Theta}^{[1..i-1]} + \vartheta^{\Theta}(i) \end{cases} \quad \text{with} \quad \begin{cases} \vartheta^{\Theta}(1) = \mu^{\Theta}(1) \\ \vartheta^{\Theta}(i+1) = \vartheta^{\Theta}(i) \cdot \sigma_{\star}^{\Theta}(i) \cdot \mu^{\Theta}(i+1) \end{cases}$$

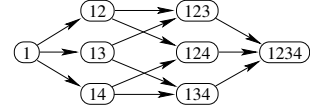
To slightly optimize this algorithm, we use the standard branch and bound technique to filter the search space. That is, we keep the currently minimal cost of leaf (a complete join) C_{min} , and stop traversing the tree as soon as $C_{\Theta}^{[1..i]} \geq C_{min}$.

As the time complexity of this join orderer clearly is $\mathcal{O}(n!)$ (and the space complexity $\mathcal{O}(n)$), it is only useful for rules with very few heads.

3.2 A \star join ordering

A second, more efficient exhaustive join ordering algorithm is based on A \star (we assume the reader is familiar with this standard algorithm [7]). The basic algorithm maintains a pool of partial joins \mathcal{J}_{Θ}^k (initially a single, empty join \mathcal{J}_{Θ}^0). In each iteration, the most promising partial join \mathcal{J}_{Θ}^k is heuristically selected, and gives rise to $n - k$ new partial joins, one for each remaining join partner. Clearly, this way the worst-case time and space complexity remains $\mathcal{O}(n!)$.

However, suppose the two partial joins (123) and (132) have the same cost, then expanding them both is pointless. We therefore use a standard A \star optimization where a *closed set* of already expanded joins is kept, where we treat joins



such as (123) and (132) as identical. Essentially, this reduces to problem to finding a shortest path in a DAG such as illustrated to the right (for $n = 3$). The worst-case time and space complexities are thus reduced to $\mathcal{O}(n \cdot 2^n)$ and $\mathcal{O}(2^n)$ respectively.

For a given set of remaining, not-yet-joined partners, the A \star algorithm requires a heuristical lower bound on the estimated cost of computing the remainder of the join. This heuristic must be *admissible*, that is, it may never exceed the actual remaining cost estimate given by the cost formula (1). Suppose a partial join has length k . Let Θ be any join order starting with the k already fixed partners. Then the actual cost (1) of joining the remaining partners X in that order is of the form:

$$C(X) = |\mathcal{J}_{\Theta}^k| \cdot \sum_{i=1}^{n-k} \left(\left(\prod_{j=1}^{i-1} \mu^{\Theta}(k+j) \cdot \sigma_{\star}^{\Theta}(k+j) \right) \cdot \mu^{\Theta}(k+i) \right) \quad (2)$$

Because $|\mathcal{J}_{\Theta}^k|$ depends only on the already fixed partial join, the problem is reduced to finding a heuristic H that is an efficiently computable tight lower bound on the remaining sum. The following two concepts will be crucial for this:

- The *minimal multiplicity* μ^{min} of a head conjunct is heuristically estimated as the expected number of constraints that satisfy the (implicit) a-priori equality guards on the conjunct's arguments, assuming all shared variables are given (or in other words: assuming it is looked up as the last partner in the join order, using optimal equality indexing).
- The *maximal (a-posteriori) selectivity* σ_{\star}^{max} is heuristically estimated as the expected probability that the a-posteriori guards hold for a given constraint matching the a-priori guards, again assuming all these guards can be tested. The *maximal* selectivity is actually the *minimal* probability of entailment.
- We further define $\gamma^{min} = \mu^{min} \cdot \sigma_{\star}^{max}$ for each partner, intuitively the *minimal cardinality* of the set of constraints matching a head conjunct.

For each partner, these estimates only have to be computed once (cf. [2, 3, 13] for a detailed description on estimating multiplicities and selectivities).

Original heuristic Let C_X be the sequence of γ^{min} values of the partners in X , sorted from small to large, and M_X^0 and S_X^0 the sequences of μ^{min} and σ_\star^{max} values of the corresponding heads, that is: $\forall i : C_X[i] = M_X^0[i] \cdot S_X^0[i]$. To compute the C_X , M_X^0 , and S_X^0 sequences, it suffices to sort all conjuncts of a given head once.

Using this notation, the heuristic proposed by [2] and [13] is given by:

$$H_0(X) = \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} C_X[j] \right) \cdot M_X^0[i] \quad (3)$$

Unfortunately, this heuristic is inadmissible. The premise of this heuristic is that, by sorting the γ^{min} values, the sum in (3) is minimized. To show that this premise does not hold, suppose we swap the elements α and β of sequences C , M_0 and S_0 ($1 \leq \alpha < \beta \leq n - k$). The sum then becomes:

$$H'_0(X) = \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} C'_X[j] \right) \cdot M_X^0[i]$$

Clearly, the terms for $i < \alpha$ and $i > \beta$ remain unchanged after swapping, and

$$\begin{aligned} H_0(X) - H'_0(X) = & \left(\prod_{j=1}^{\alpha-1} C_X[j] \right) \cdot \left(M_X^0[\alpha] - M_X^0[\beta] \right. \\ & + (C_X[\alpha] - C_X[\beta]) \cdot (M_X^0[\alpha+1] + \dots) \\ & \left. + (S_X^0[\alpha] - S_X^0[\beta]) \cdot M_X^0[\alpha] \cdot M_X^0[\beta] \cdot \prod_{k=\alpha+1}^{\beta-1} C_X[k] \right) \end{aligned}$$

If the heuristics' premise were correct, then $H_0(X) \leq H'_0(X)$. But then not only must $C_X[\alpha] \leq C_X[\beta]$, but also $M_X^0[\alpha] \leq M_X^0[\beta]$ and $S_X^0[\alpha] \leq S_X^0[\beta]$. In general, however, sorting the products of M^0 and S^0 does not guarantee that the sequences themselves are sorted. A counter-example is easily obtained by $C = [1, 3]$, $M^0 = [2, 15]$ and $S^0 = [0.5, 0.2]$, where the latter is unsorted.

Correct heuristics A first correct underestimate is derived as follows. Observe that the i th term in the sum of the actual cost (2) is given by a product of i σ_\star^Θ and $i + 1$ μ^Θ values. A correct lower bound for the i th term is thus the product of the i smallest σ_\star^{max} , and the $i + 1$ smallest μ^{min} values. First, we therefore sort both the σ_\star^{max} and the μ^{min} values of all occurrences in X in two sequences S_X and M_X . Again, in practice, two global S and M lists are computed, from which S_X and M_X are readily derived. The heuristic H_1 is given by:

$$H_1(X) = \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} M_X[j] \cdot S_X[j] \right) \cdot M_X[i]$$

This heuristic only coincides with H_0 if the sequences M_X^0 and S_X^0 happen to be sorted, which as shown earlier is not always the case.

In H_1 , we observe that $M[i]$ and $S[i]$ generally do not originate from the same occurrence, while in the actual cost, the $\mu^\Theta(i) \cdot \sigma_\star^\Theta(i)$ factors do belong to the same occurrence. An alternative underestimate is thus based on a sequence C_X defined as before, and the smallest minimal multiplicity of all occurrences in X , i.e. $M_X[1]$:

$$H_2(X) = \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} C_X[j] \right) \cdot M_X[1]$$

Again, each term clearly underestimates the corresponding term in the actual cost. The difference with H_0 is that instead of multiplying with $M_X^0(i)$, each term is multiplied with $M_X[1]$, a trivially safe (yet possibly very poor) underestimate.

When comparing H_1 and H_2 , there is no clear winner. Obviously

$$H_1(X) = M_X[1] \cdot \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] \right)$$

and therefore

$$H_1(X) - H_2(X) = M_X[1] \cdot \sum_{i=1}^{n-k} \left(\prod_{j=1}^{i-1} S_X[j] \cdot M_X[j+1] - \prod_{j=1}^{i-1} C_X[j] \right)$$

Neither of the above heuristics is thus superior in itself. The heuristic currently used by JCHR2 is therefore simply

$$H_3(X) = \max(H_1(X), H_2(X))$$

It provides fairly tight lower bounds, while still remaining admissible and efficiently computable. We need only to compute three sorted sequences M , S , and C containing the values for all join partners of a head once. Using these sequences, computing the heuristic H_3 has a reasonable runtime cost linear in the number of remaining partners.

4 Randomized Algorithms

While our A^* join orderer scales reasonably well, it remains an exponential algorithm. In fact, as join ordering is NP complete [9], any exhaustive algorithm is bound to be infeasible in general. For really large heads (currently $n > 10$ in JCHR2), we must therefore fall back to randomized algorithms that compute reasonable—though not necessarily optimal—join orders in reasonable time.

Our current implementation uses local search algorithms inspired by the ‘iterative improvement’ algorithm of [17]. This algorithm is essentially a random-restart hill climbing algorithm, but we extended it to a random-restart beam search algorithm. Starting from some initial join ordering (chosen either randomly, or using some greedy join ordering algorithm), this join order is incrementally improved, by randomly generating small changes (e.g. swapping two partners; cf. [17]), and updating the current order each time such a change results in a cost improvement. In the beam search variant, a fixed set of the b best join orders is kept instead of just the one. Once a threshold of subsequent unsuccessful local changes is met, the algorithm repeats the same process with a different (pseudo-random) initial join ordering. The algorithm ends, once some stop criterium is met (currently either a fixed number of (unsuccessful) restarts, or some timeout polynomial in n).

Space limitations prohibit a more detailed description. In any case, more experimentation is needed to tune the many parameters of the algorithm (cf. Section 8).

5 On the KBZ Algorithm

In [2, 3, 13], an $\mathcal{O}(n \log n)$ algorithm is presented for the join ordering of a common, specific type of rule heads (those with acyclic join graphs to precise; cf. [2, 3, 13]). It is currently not yet implemented in JCHR2. Still, our preliminary analysis already revealed the following two issues, relevant to anyone who wants to implement it:

1. In [2, 3, 13], wrongfully call their algorithm a KBZ algorithm, and accredit it to [10]. The algorithm they actually describe is the IK algorithm, which was first applied to join ordering by [9]. The real KBZ algorithm of [10] further improves on the IK algorithm. When applied to our problem, it computes the join order for all n active constraints of a given head in $\mathcal{O}(n^2)$ time instead of $\mathcal{O}(n^2 \log n)$.

2. We believe the version of the IK algorithm described by [10], and subsequently copied by [2, 3, 13], is not correct. It uses a step where chains of nodes are merged, but the problem is that these chains may not be sorted. The normalisation it performs at the root of the merged chains does not resolve this. The true IK algorithm correctly normalises (sorts) both chains before merging [9].

6 Related work

Join ordering received considerable attention in database research [9, 10, 18], too much to cover here. We refer e.g. to [16] for a good survey on randomized join ordering algorithms.

CHR implementations currently use ad-hoc join ordering heuristics, typically based on those described by [5, 8]. We refer to [2, 3, 13] for a detailed discussion why the cost model underlying these heuristics is flawed. The algorithms used to minimize the estimated cost, moreover, are mostly crude and ineffective. Both HALCHR [5, 8] and the initial JCHR system [21] use a linear, greedy algorithm, that often leads to suboptimal results. The K.U.Leuven CHR system uses a naive A*-based algorithm (i.e. without closed set), but only if $n < 6$. For larger multi-headed rules, the partners are simply joined left-to-right. The CCHR [23] system uses a straightforward branch-and-bound optimization algorithm for $n \leq 8$; for larger heads it simply generates 40,000 random join orders and keeps the best. Clearly, given the importance of join ordering, settling for such ad-hoc algorithms cannot be excused.

7 Conclusions

Join ordering is fundamental for the optimal runtime complexity of CHR programs. Nevertheless, both the heuristics (cf. [2, 3, 13]) and algorithms (cf. Section 6) used by current systems are very ad-hoc. The first issue was addressed by [2, 3, 13], the latter in this paper. Practice shows that, unlike claims to the contrary in [5, 8], CHR programs do frequently contain complex multi-headed rules ([2, 3, 13] provide examples). The careful design and implementation of adequate join ordering algorithms is therefore indispensable. We outlined how to translate the theoretical model of [2, 3, 13] into efficient, flexible join ordering algorithms. For JCHR2, we implemented three join orderers, based on branch-and-bound, A*, and local search respectively, each used for increasingly larger rule heads. We also listed some considerations on implementing a more efficient poly-time KBZ algorithm (as first proposed in [2, 3, 13]). The contributions reported in this paper are relevant to anyone who wants to implement join ordering (based on [2, 3, 13]).

8 Future work

The current combination used by JCHR2 seems to work well in practice. Still, we only have scratched the surface, and more experimentation is required to determine:

1. whether the assumptions made by the cost function of [2, 3, 13], and the heuristics used to estimate it, are indeed appropriate;
2. the optimal parameters for the local search algorithm (starting points, local moves, beam size, stopping criteria, etc.). Also, alternative randomised algorithms (genetic algorithms, simulated annealing, etc.) could be considered [16].

Many more issues must be further investigated: first-few answers (cf. [1, 2, 13]), join strategies besides nested-loop joins, a-priori guards besides equality, etc.

The most important open problem though is that, short of reliable estimates for e.g. cardinalities and selectivities, static join ordering frequently will always remain error-prone. To mend this shortcoming, we proposed annotations that allow the

user to specify cardinalities and selectivities in [20]. While these annotations help, they rely on the programmer to supply sufficient and correct information. The only really efficacious solution is dynamic join ordering (cf. [2, 3, 13, 19]).

References

1. R. J. Bayardo, Jr. and D. P. Miranker. Processing queries for first-few answers. In *CIKM '96: Proc. fifth intl. Conf. Inf. and Knowl. Mgmt.*, pages 45–52. ACM, 1996.
2. L. De Koninck. *Execution Control for Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, Nov. 2008.
3. L. De Koninck and J. Sneyers. Join ordering for Constraint Handling Rules. In Djelloul et al. [4], pages 107–121.
4. K. Djelloul, G. J. Duck, and M. Sulzmann, editors. *CHR '07: Proc. 4th Workshop on Constraint Handling Rules*, Sept. 2007.
5. G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Australia, Dec. 2005.
6. T. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to “a formal basis for the heuristic determination of minimum cost paths”. *SIGART Bull.*, (37):28–29, 1972.
8. C. Holzbaaur, M. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. volume 5(4–5) of *Theory and Practice of Logic Programming*, pages 503–531. Cambridge University Press, July 2005.
9. T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.
10. R. Krishnamurthy, H. Borral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB '86: Proc. 12th Intl. Conf. on Very Large Data Bases*, pages 128–137, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.
11. T. Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, June 2005.
12. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *CHR '04, Selected Contributions*, pages 8–12, May 2004.
13. J. Sneyers. *Optimizing Compilation and Computational Complexity of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Belgium, Nov. 2008.
14. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS*, 31(2), Feb. 2009.
15. J. Sneyers, P. Van Weert, T. Schrijvers, and L. De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *TPLP*, 10(1):1–47, 2010.
16. M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, 1997.
17. A. Swami and A. Gupta. Optimization of large join queries. *SIGMOD Rec.*, 17(3):8–17, 1988.
18. A. N. Swami and B. R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 345–354, Washington, DC, USA, 1993. IEEE Computer Society.
19. P. Van Weert. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, 2010. To appear.
20. P. Van Weert, L. De Koninck, and J. Sneyers. A proposal for a next generation of CHR. In *CHR '09*, pages 77–93, July 2009.
21. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *CHR '05*, pages 47–62, 2005.
22. P. Van Weert, P. Wuille, T. Schrijvers, and B. Demoen. CHR for imperative host languages. volume 5388 of *LNAI*, pages 161–212. Springer, Dec. 2008.
23. P. Wuille, T. Schrijvers, and B. Demoen. CCHR: the fastest CHR implementation, in C. In Djelloul et al. [4], pages 123–137.

The Viterbi Algorithm expressed in Constraint Handling Rules^{*}

Henning Christiansen, Christian Theil Have, Ole Torp Lassen, and Matthieu Petit

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
{henning, cth, ot1, petit}@ruc.dk

Abstract. The Viterbi algorithm is a classical example of a dynamic programming algorithm, in which pruning reduces the search space drastically, so that an otherwise exponential time complexity is reduced to linearity. The central steps of the algorithm, expansion and pruning, can be expressed in a concise and clear way in CHR, but additional control is needed in order to obtain the desired time complexity. It is shown how auxiliary constraints, called *trigger* constraints, can be applied to fine-tune the order of CHR rule applications in order to reach this goal. It is indicated how properties such as confluence can be useful for showing such optimized programs correct.

1 Introduction

Hidden Markov Models (HMMs) are probabilistic finite state machines that for each transition emits a symbol from a finite alphabet, also by probabilistic choice. HMMs are commonly used for modeling and analysis of, e.g., biological sequence data and for speech recognition; see, e.g., [8, 12]. Given a specific HMM and an observed sequence over the alphabet, *prediction* means to find the most probable path, i.e., sequences of states, by means of which the sequence may have been produced; such a path is called a *Viterbi path*. Informally speaking, a Viterbi path represents the most feasible interpretation or explanation of the given sequence; in the biological case, the sequence may be DNA and the Viterbi path indicates the most believable shifts between coding and non-coding regions, and perhaps details concerning introns and exons [8].

HMMs owe much of their popularity to the existence of efficient algorithms for training and, as we consider here, prediction in terms of the classical Viterbi algorithm [17]. It is a dynamic programming algorithm that gradually extends optimal paths so they cover a longer and longer prefix of the sequence, and eventually the entire sequence. The algorithm keeps track of one optimal path ending in each state s for the sub-sequence seen so far; call this set of partial paths Σ . Any non-optimal path is discarded. In the next step, a new set of optimal paths is found among the possible extensions of any $\sigma \in \Sigma$ with one more state. The time complexity is $\mathcal{O}(n \cdot k^2)$ and the space complexity is $\mathcal{O}(n \cdot k)$ where n is the sequence length and k the number of states.

In this paper, we investigate how well the Viterbi algorithm can be expressed in CHR, considering both conciseness and efficiency. CHR, or Constraint Handling Rules [9, 10], was introduced as a declarative language for writing constraint solvers, but has shown to be very useful for a variety of automated reasoning tasks, and attempts have been made to use it as a general language for describing algorithms.

^{*} This work is part of the project “Logic-statistic modeling and analysis of biological sequence data” supported by the NABIIT program under the Danish Strategic Research Council. We thank also anonymous reviewers for suggestions to improve our solutions.

We show that the fundamental steps in the Viterbi algorithm can be exposed very clearly in CHR, but to reach the optimal time complexity, we need to introduce some techniques. We suggest to use *trigger* constraints, by means of which a program’s operational behavior can be fine-tuned. For confluent programs, this can be analyzed in a systematic way, and as a more general case, we put forward informally the notion of “relative confluence” based on a more flexible state equivalence (as compared with the usual logical equivalence of states). However, in order to reach the optimal time complexity, we need to make additional transformations that reflect the underlying CHR system’s search and matching. This may be less satisfactory from the point of view of declarative programming, but may inspire to the development of new automatic analyses and transformations to be included in CHR implementations.

2 A concise Viterbi-like algorithm in CHR

The fundamental parts of the Viterbi algorithm can be expressed in CHR as shown in fig. 1; the specific HMM is encoded as a set of constraints of the forms **trans**(q_1, q_2, p_1) and **emit**(q_3, ℓ, p_2), where p_1 is the probability to transit from state q_1 to q_2 , and p_2 is the probability to emit the letter ℓ in state q_3 . For simplicity and wlog, we assume a unique initial state, consistently called **q0**, and that any state serves as a final state. The intuitive meaning of a constraint **path**(E, q, p, π) is that there exists a partial path starting in **q0** and ending in q with probability p , and E is the remaining part of the sequence that needs to be analyzed in order to complete a full path; for ease of programming, the argument π represents this partial path in reversed order. The initial query should be stated as “:- *HMM*, **path**($L, \mathbf{q0}, 1, []$)”

```
:- chr_constraint path/4, trans/3, emit/3.

expand @ trans(Q,Q1,PT), emit(Q,L,PE), path([L|Ls],Q,P,PathRev) ==>
  P1 is P*PT*PE, path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P2 | true.
```

Fig. 1. A naive Viterbi-like algorithm in CHR

where *HMM* is an encoding of the particular HMM and L a sequence to be analyzed.

Termination follows from the fact that the **expand** rule always reduces the length of the first argument in the involved **path** constraint. A correctness proof, which is left out due to space limitations, can be made by induction showing that **prune** will eventually remove any non-optimal **path**, but always leaves an optimal one for any prefix of the sequence, and that **expand** produces all possible extensions of an optimal **path** (for any proper prefix). This proof does not need any assumptions about the order in which the rules are applied.¹

Let us informally analyze the time complexity of this program. For simplicity we count only the number of constraints that are created during the derivation; for a detailed analysis, we may refer to the methods of [11, 6].² Assuming a naive, nondeterministic semantics, we may observe derivations that are exponential in the length of the sequence to be analyzed; this is the case when, e.g., **expand** is applied as long as possible, before any application of **prune**. Our benchmarks (see appendix) confirm the exponential behaviour; interestingly, when swapping the order of the

¹ Notice that the program of fig. 1 is not confluent, although intuitively very close; we consider this in more detail in section 4 below.

² Our simplified time complexity measure abstracts away the cost of search and matching performed by the CHR system.

rules (i.e., **prune** first), our tests seem to indicate³ a time complexity of $\mathcal{O}(n^4)$, although we cannot present a proof for this hypothesis. This is of course far too slow for any interesting application, and also unsatisfactory as it is known that the algorithm can run in linear time when written in an imperative language.

3 Fine-grained control by trigger constraints

Linear time complexity requires an optimal interleaving of the **expand** and **prune** rules, so that any **path** constraint, which will be **pruned** sooner or later, is not **expanded**. We can sketch a class of derivations of linear size by the pseudo-code shown in fig. 2. As an attempt to obtain a similar flow of control in CHR, and we

```
seq:= L;
while seq ≠ [] do
  1) apply expand as long as possible to constraints of form
      path(seq,q,p,π), for any q, p and π;
  2) apply prune as long as possible;
  3) seq:= tail(seq);
```

Fig. 2. Pseudo-code for optimal control.

introduce what we call *trigger constraints* by means of which we can control the detailed procedural semantics of the underlying implementation. Fig. 3 shows an adaptation of the previous version with trigger constraints. The initial query should be stated as “:- *HMM*, **path**(L,q0,1,[]),**trigger**(L)”. During execution, the

```
:- chr_constraint path/4, trans/3, emit/3, trigger/1.

expand @ trans(Q,Q1,PT), emit(Q,L,PE),
      path([L|Ls],Q,P,PathRev), trigger([L|Ls]) ==>
      P1 is P*PT*PE, path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P2 | true.

step @ trigger([_|Ls]) <=> trigger(Ls).
```

Fig. 3. Viterbi with trigger constraints, version 1.

trigger constraint will refer to decreasing remainders of the sequence, and for each such iteration provide the relevant applications of **expand** and **prune**. This preserves the logical meaning of the original program, since 1) the trigger constraints are added only in the head of the original rules, 2) new rules concerning triggers only, e.g., the **step** rule, do not unify any arguments, and 3) no derivation is stopped in a state where the original program would be able to extend the derivation. Notice that such a proof would need to refer to the operational semantics of the underlying implementation as well as to the order of the constraints in the initial query.

We sketch an analysis of the time complexity based on the operational semantics of standard CHR implementations. No rule will execute before **trigger**(L) is called in the initial query, and when this happens, **expand** will apply as long as possible for any **path** constraint referring to L similarly to what is expressed in line 1 in fig. 2, however, interleaved with **prune** (line 2). When this phase is done, the constraint **trigger**(L) reaches the **step** rule and mutates into **trigger**(tail(L)) and the process repeats for tail(L), and so on, a thus leading to derivations of linear length, as the number of steps in each such iteration is independent of the sequence length.

³ This and other estimates for time complexity are made by inspecting higher order differences for the measured runtimes.

For a fully connected HMM with k states, k^2 new `path` constraints are created in each iteration, so the length of the entire derivation becomes $\mathcal{O}(n \cdot k^2)$. However, the actual time complexity may become higher as we did not count the time for matching of list arguments in the `expand` and `prune` rules, which may, in the worst case, add another factor n to the time complexity, thus $\mathcal{O}(n^2 \cdot k^2)$. In fact, our runtime tests shown in the appendix suggest $\mathcal{O}(n^3)$ for both for a randomly generated sequence and a worst case sequence that repeats a single letter. The latter implies that the comparison of two list arguments always traverses the shortest sequence to the very end; the benchmarks indicate a huge constants factor between the two.

In order to reduce time for matching, we may add a new argument representing the length of the sequence to `path` constraints and let the trigger depend on this length only; the resulting program is shown in fig. 4. Assuming an implementation

```
:- chr_constraint path/5, trans/3, emit/3, trigger/1.

expand @ trans(Q,Q1,PT), emit(Q,L,PE),
        path(N,[L|Ls],Q,P,PathRev), trigger(N) ==>
        P1 is P*PT*PE, N1 is N-1, path(N1,Ls,Q1,P1,[Q1|PathRev]).

prune @ path(N,_,Q,P1,_) \ path(N,_,Q,P2,_) <=> P1 >= P2 | true.

step @ trigger(N) <=> N > 0 | N1 is N-1, trigger(N1).
```

Fig. 4. Viterbi with trigger constraints, version 2, with length arguments.

that applies a suitable indexing on the first argument, we would expect this to lead to a linear algorithm in the length of the sequence. However, benchmarks indicate worst case and average complexity of $\mathcal{O}(n^2)$, which we may hypothesize relates to a non-optimal search for `path` constraints. Swapping the `expand` and `prune` rules only changed the figures with a few percent.

To finally overcome these problems and to reach the theoretically best time complexity for Viterbi in CHR, we needed to add explicit passive declarations⁴ and additional code to remove non-current path constraints; our experiments showed that both additions were necessary. Such a program is shown in fig. 5. We expect that

```
:- chr_constraint path/5, trans/3, emit/3, trigger/1, zap/1.

expand @ trans(Q,Q1,PT) # Id1, emit(Q,L,PE) # Id2,
        path(N,[L|Ls],Q,P,PathRev) # Id3, trigger(N) ==>
        P1 is P*PT*PE, N1 is N-1, path(N1,Ls,Q1,P1,[Q1|PathRev])
        pragma passive(Id1), passive(Id2), passive(Id3).

prune @ path(N,_,Q,P1,_) \ path(N,_,Q,P2,_) <=> P1 >= P2 | true.

step @ trigger(N) <=> N > 0 | zap(N), N1 is N-1, trigger(N1).
zap(N) \ path(N,_,_,_,_) # Id <=> true pragma passive(Id).
zap(_) <=> true.
```

Fig. 5. A linear time Viterbi algorithm in CHR; passive declarations and removal of non-current path constraints.

a detailed analysis can prove linear complexity. Indeed, our benchmarks indicate that it does stay linear until sequence lengths of more than 10,000 and increases significantly from around 20,000 and upwards. While a sequence of length 10,000 can be analyzed in 30 sec., it takes 45 minutes for length 100,000. We expect that this is related to the memory being exhausted due to extreme stack sizes.

⁴ Passive declarations are a low-level device that suppresses certain firings of rules; for details, see, e.g., a manual for any of the major CHR versions in Prolog.

4 Conclusion: Methodological considerations, future and related work

We have shown an implementation of the Viterbi algorithm in CHR, starting from an abstract and concise specification expansion and pruning. Systematic extensions by triggers and other techniques lead to an implementation with ideal time complexity. The Viterbi algorithm represents a larger class of dynamic programming algorithms for which we believe that our techniques can be applied.

The naive program of fig. 1 is not confluent due to the fact that, when two paths exist for the same sub-sequence and with the same probability, `prune` may nondeterministically get rid of an arbitrary one of them, leading to different new states that are not logically equivalent. However, it satisfies a requirement that we may call *relative confluence* based on an application specific state equivalence relation. If, for example, two states differs only by the exchange of `path(L, p, q, π)` for `path(L, p, q, π')`, we consider these states equivalent.

Our plans for future work include the formalization of relative confluence and to generalize known results for confluent programs [10, 1] accordingly. We believe that this can be very useful as many interesting non-confluent programs are relatively confluent. For confluent programs, it is possible to show as a general result, that the addition of trigger constraints – satisfying the requirements noticed above for the program of fig. 3 – preserves the logical meaning of the program as concerns its original constraints.

Additional optimizations were needed in order to obtain the best time complexity, based on detailed knowledge about the underlying machinery. Our experience in doing the exercise for the Viterbi algorithm may inspire to more advanced, automatic analyses and transformations being applied in CHR implementations in order to promote declarative programming with competitive execution times. We may also consider the ideal of a true separation of logic and control, so that we might do with the naive program of fig. 1, complemented by an additional control specification, which may resemble our abstract algorithm in fig. 2.

An attempt to obtain such a separation have been done by rule priorities [14]. The priority of each rule is expressed in terms of an arithmetic expression referring to variables in the head of the rules. While the control mechanism appears as decorations to the rules, rather than infiltrating the code as our triggers do, it is also clear that the rules need to be designed in the first place so that the rule heads actually contain the necessary information. For the Viterbi algorithm it seems obvious that the sequence length needs to be present in order to express relevant priorities. We have not tried to express the Viterbi algorithm in CHR with rule priorities, but it seems to require advanced algebraic skills to encode the desired control pattern.

There has been other work studying CHR for expressing algorithms [13]. We will emphasize [16] that gives a detailed analysis of how Dijkstra’s shortest path algorithm [7] can be implemented in CHR; specifically, the authors studied the use of priority queues. Similar techniques have been employed by [3] for probabilistic abductive logic programming in CHR, by [2] for soft constraints and by [15] to express imperative control constructs in CHR. We have not seen any earlier, systematic approach for adding detailed procedural control to confluent (or relatively confluent) programs in order to get the best out of the pruning rules. The Viterbi algorithm has been formulated as a constraint problem by [4, 5], but not in CHR.

Finally, we notice that CHR is suited for describing the fundamental steps of interesting algorithms, but it is difficult to consider it as a serious implementation language at present. While the theoretically best complexity often can be reached in CHR, there is typically a huge constant factor due to the overhead in the underlying CHR and Prolog runtime systems. For the Viterbi algorithm, it is thought provoking that it can be implemented very efficiently by a handful of lines of imperative code.

References

1. Abdennadher, S., Frühwirth, T., Meuss, H.: On confluence of Constraint Handling Rules. In: Freuder, E.C. (ed.) CP. Lecture Notes in Computer Science, vol. 1118, pp. 1–15. Springer (1996)
2. Bistarelli, S., Frühwirth, T., Marte, M., Rossi, F.: Soft constraint propagation and solving in constraint handling rules. *Computational Intelligence* 20(2), 287–307 (2004)
3. Christiansen, H.: Implementing probabilistic abductive logic programming with Constraint Handling Rules. In: Schrijvers, T., Frühwirth, T. (eds.) Constraint Handling Rules, Lecture Notes in Computer Science, vol. 5388, pp. 85–118. Springer (2008)
4. Christiansen, H., Have, C.T., Lassen, O.T., Petit, M.: Inference with Constrained Hidden Markov Models in PRISM (2010), accepted for ICLP 2010, Int’l Conf. on Logic Programming; to appear in *Theory and Practice of Logic Programming*
5. Costa, V.S., Page, D., Cussens, J.: Clp(n): Constraint logic programming for probabilistic knowledge. In: Raedt, L.D., Frasconi, P., Kersting, K., Muggleton, S. (eds.) Probabilistic Inductive Logic Programming. Lecture Notes in Computer Science, vol. 4911, pp. 156–188. Springer (2008)
6. De Koninck, L.: Execution Control for Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Belgium (Nov 2008),
7. Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959),
8. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press (1999)
9. Frühwirth, T.: Theory and practice of Constraint Handling Rules. *Journal of Logic Programming* 37(1-3), 95–138 (1998)
10. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (Aug 2009),
11. Ganzinger, H., McAllester, D.A.: A new meta-complexity theorem for bottom-up logic programs. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR. Lecture Notes in Computer Science, vol. 2083, pp. 514–528. Springer (2001)
12. Jurafsky, D., Martin, J.H.: Speech and Language Processing (2nd Edition) (Prentice Hall Series in Artificial Intelligence). Prentice Hall, 2 edn. (2008)
13. Koninck, L.D., Schrijvers, T., Demoen, B.: The correspondence between the logical algorithms language and CHR. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 209–223. Springer (2007)
14. Koninck, L.D., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: Leuschel, M., Podelski, A. (eds.) PDP. pp. 25–36. ACM (2007)
15. Meister, M.: Advances in Constraint Handling Rules. Ph.D. thesis, Universität Ulm, Germany (2008)
16. Sneyers, J., Schrijvers, T., Demoen, B.: Dijkstra’s algorithm with Fibonacci heaps: An executable description in CHR. In: Fink, M., Tompits, H., Woltran, S. (eds.) WLP. INFSYS Research Report, vol. 1843-06-02, pp. 182–191. Technische Universität Wien, Austria (2006)
17. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13, 260–269 (April 1967)

Appendix: Benchmarks

The different variants of the Viterbi algorithm have been tested for a fixed, fully connected HMM with 4 states (plus a start state that cannot be re-entered) and an emission alphabet of 4 letters. We have measured runtimes as functions of the sequence lengths. In most cases, we test on a randomly generated sequence, considering it as “typical” or “average”. Tests were made with SICStus Prolog 4.0.4 on a Macintosh 2.4 GHz Intel Core 2 Duo with 4GB RAM, and runtimes have been measured using SICStus Prolog’s `statistics(runtime, ...)` device that ignores any time spent on garbage collection and other memory management tasks. Runtimes below 10 seconds were taken as average of 10 runs, whereas higher ones were measured by a single run. Space complexity was not considered.

Fig. 6 shows runtimes for the naive algorithm (fig. 1) for the two alternative ordering of its rules. The top curve to the left, for the `expand` rule first, confirms

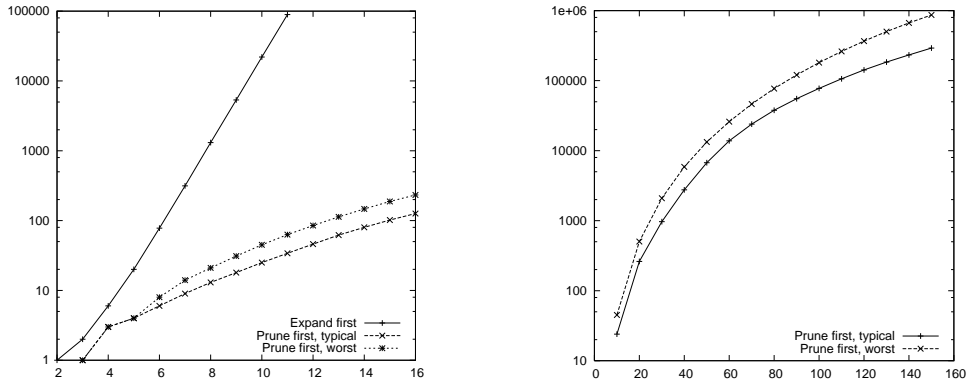


Fig. 6. Naive algorithm with different rule orders; x -axis n , y -axis ms log-scaled.

our expectation of exponential complexity. Swapping the rules so that `prune` comes first, reduces the complexity drastically. We have measured for random sequences (the “typical”) plus the worst case for this algorithm, which are sequences that repeat a single letter. The right part shows the `prune` first version for the two sorts of sequences for n up to 150. An inspection of higher order differences made from the actual figures indicates that $O(n^4)$ is a reasonable hypothesis for both typical and worst case; the worst case is about 3 times slower than the typical for $n = 150$.

Fig. 7 shows runtimes for the versions that use trigger constraints. The two top curves to the left shows typical and worst case for the version where triggers use lists, cf. fig. 3. In both cases, differences seem to indicate $O(n^3)$; for $n = 150$ there is a factor 24 between typical and worst case. The measurements when triggers use the list length, as opposed to the actual list, is shown as the lowest curve in the left part and continues as the top curve to the right. Differences suggest $O(n^2)$; the typical and worst case used above provides the same runtimes and swapping the `expand` and `prune` rules changes only a few percent. Finally, the right part of fig. 7 shows

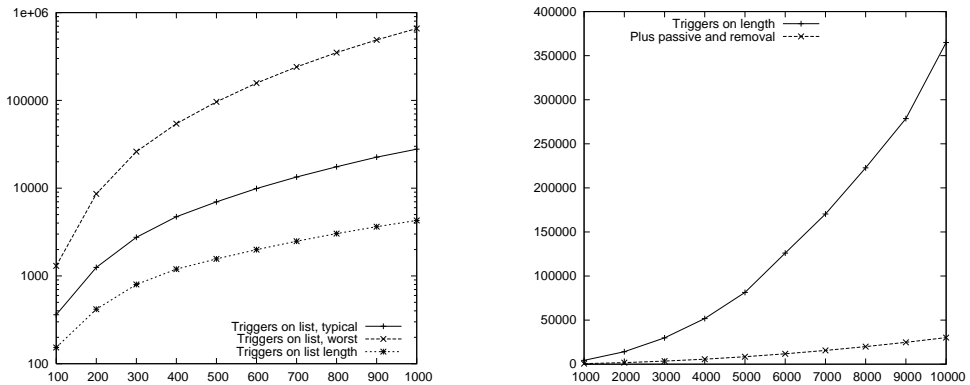


Fig. 7. Algorithm with triggers based on sequences and lengths; x -axes n , y -axes ms log-scaled in left part, linear in the right part.

also runtimes for the fully optimized version of fig. 5 with passive declarations and removal of non-current path *constraints*. It stays linear up to above $n = 10,000$, and for larger values, 20,000 and upwards (not shown), the time grows higher, most likely because memory begins to be exhausted due to stack sizes. For $n = 10,000$, the time is about half a minute and for $n = 100,000$, 45 minutes.

Result-directed CHR Execution

Jon Sneyers

Dept. of Computer Science, K.U.Leuven, Belgium
jon.sneyers@cs.kuleuven.be

Abstract. The traditional execution mode of CHR is bottom-up, that is, given a goal, the result is computed by exhaustively applying rules. This paper proposes a result-directed execution mode for CHR, to be used when both the goal and the result are known, and the task is to find all corresponding derivations. Result-directed execution is needed in the context of CHRiSM, a probabilistic extension of CHR in which goals typically have a large number of possible results. The performance of result-directed execution is greatly improved by adding early-fail rules.

1 Introduction

Traditionally, execution of Constraint Handling Rules [1, 2] works as follows. Starting from an initial state σ_i (also called the goal or query), execution consists of applying the rules of the program exhaustively according to some execution strategy. If the program terminates, some final state σ_f is reached in which no more rules are applicable (called the result or answer). The traditional execution mode is, in a sense, a blind bottom-up computation. Hence it is often desirable that programs are (observably) confluent [3], i.e., every goal has a *unique* result. More precisely, if a program is non-confluent w.r.t. the very nondeterministic theoretical operational semantics ω_t , then it should at least be confluent w.r.t. the strategy class [4] for which it was written, e.g. the less nondeterministic refined strategy class [5].

CHRiSM [6] is a rule-based probabilistic programming language based on an implementation of CHR(PRISM); PRISM is a probabilistic extension of Prolog [7]. In particular, the current CHRiSM implementation is based on the K.U.Leuven CHR system in B-Prolog. In the context of CHRiSM, most programs are non-confluent because the result of a goal depends on probabilistic choices made during the computation. The traditional execution mode corresponds to sampling, that is, given a goal, one of many possible results is returned, with a probability that depends on the probabilities of the choices that were made.

Traditional execution works well for sampling CHRiSM programs. However, there are other probabilistic inference tasks. Given a goal G and a result R , one may want to know the probability that R is returned given G . Or one may want to find the most likely (Viterbi) explanation, that is, the most likely sequence of probabilistic choices that lead to R . Or one may want to perform a learning algorithm to estimate the probability distributions of the choices in order to maximize the likelihood of given observations. For all these tasks, the basic challenge is to find all explanations of a result given a goal, that is, all computation paths from G to R .

PRISM is designed to find explanations for Prolog goals. It does this essentially by making the probabilistic choices backtrackable (during sampling they are committed-choice) and using a failure driven loop to find all explanations. CHRiSM computations can be wrapped in Prolog as follows:

```
computation(Goal,Result) :-  
    metacall(Goal),  
    get_chr_store(Store),  
    compare_multiset(Result,Store).
```

This mechanism works, but it is essentially a generate-and-test approach, which uses traditional execution to produce *all possible* results for `Goal`. This is horribly inefficient. We need a *result-directed* execution mode.

2 Result-directed Execution

When executing CHR in a result-directed way, the aim is not to compute the result R for a given goal G , but rather to find derivations from G to R , given both G and R . Using the notation and terminology of CHRiSM, we are looking for an *explanation* for the (full) *observation* $G \Leftarrow R$. Sometimes only a *partial* observation $G \Leftarrow P$ is given, which means that the result is only partially known, that is, P is a (multiset) subset of the actual result.

Result-directed execution is needed in CHRiSM to find all explanations of an observation, which is a necessary step for probability computation and for parameter learning. However, it is also a useful execution mode for CHR in general, especially in variants of CHR that involve search, like CHR^\vee [8].

Approach. The approach is as follows. We start from the naive generate-and-test approach described in the introduction: compute all results for G , and after each computation, compare the final CHR store with the desired result R , failing if there is a difference. In order to make this more efficient, we now try to do the failing as soon as possible, pruning away redundant computations. We use a source-to-source transformation of the CHR program to achieve this pruning. Three new constraints are introduced: `result/1`, `observation/1`, and `cleanup/0`. We adapt the wrapper predicate from the introduction as follows:

```
computation(Goal,Result,S) :-
    metacall((observation(S),result(Result),Goal,cleanup)),
    get_chr_store(Store),
    compare_multiset(Result,Store).
```

The argument `S` indicates the status of the observation: `full` or `partial`. If the observation is `full`, then `Result` encodes the entire final store; if the observation is `partial`, then `Result` is some subset of the final store.

We add the following rules to (the bottom of) the original program:

```
result((A,B)) <=> result(A), result(B).
cleanup \ observation(_) <=> true.
cleanup \ result(_) <=> true.
cleanup <=> true.
```

The first rule recursively splits the result conjunction in its conjuncts; the `cleanup` rules make sure that none of these new constraints are visible in the actual result. Note that we use the refined operational semantics [5].

Now we add a number of *early-fail rules* to the top of the original program.

3 Early-fail Rules

Early-fail rules detect situations in which it has become impossible to reach the desired result. The redundant further computation is pruned by failing immediately.

3.1 Never-removed Constraints

Some programs have *never-removed* constraints. These constraints do not occur in the removed part of rule heads. As a consequence, once a never-removed constraint is added, it will remain in the store until the final result is reached.

If we know that a constraint c is never-removed (and ground), we can add an early-fail rule of the following form:

```
observation(full), c( $\bar{X}$ ) ==> check(c( $\bar{X}$ )).
result(c( $\bar{X}$ )) \ check(c( $\bar{X}$ )) <=> true.
check(c( $\bar{X}$ )) <=> fail.
```

We introduce a new constraint, `check/1`, which searches for a matching `result/1` constraint and fails if no match is found. We use the refined semantics for this, in a similar way as in CHR^1 [9].

For efficiency reasons we do some flattening, as described in [10]:

```
result(c( $\bar{X}$ )) <=> result_c( $\bar{X}$ ).
cleanup \ result_c( $\square$ ) <=> true.
observation(full), c( $\bar{X}$ ) ==> check_c( $\bar{X}$ ).
result_c( $\bar{X}$ ) \ check_c( $\bar{X}$ ) <=> true.
check_c( $\bar{X}$ ) <=> fail.
```

Detecting never-removed constraints is straightforward; however, note that the above early-fail rules are only sound if the full result is known.

3.2 Partial Observations

If we only know part of the result, the above approach cannot be used. However, if we know that a never-removed constraint c has a *functional dependency* [11] between its arguments, we can add another kind of early-fail rules. We say a constraint c/n exhibits a functional dependency $K \rightsquigarrow V$ (where K and V partition the set of n argument positions) if the *key* arguments K uniquely determine the arguments V . E.g., if a program starts with the rule

$$c(\bar{X}, \bar{Y}, \square), c(\bar{X}, \bar{Z}, \square) \implies \bar{Y} = \bar{Z}.$$

then the constraint $c(\bar{X}, \bar{Y}, \square)$ has a functional dependency $\bar{X} \rightsquigarrow \bar{Y}$. In [11], a method is described to detect functional dependencies.

Now if we have a never-removed constraint $c(\bar{X}, \bar{Y}, \square)$ with a functional dependency $\bar{X} \rightsquigarrow \bar{Y}$, we can add early-fail rules of the following form:

$$c(\bar{X}, \bar{Y}, \square), \text{result_c}(\bar{X}, \bar{Z}, \square) \implies \bar{Y} \neq \bar{Z} \mid \text{fail}.$$

If a new c/n constraint is added, and the (partial) result contains a constraint with the same key, then we compare the functionally dependent arguments from both constraints, failing if they are different.

3.3 Surviving Constraints

Some constraints are not quite never-removed, but still behave in a sufficiently monotonous way in order to be used in an early-fail rule. Consider for example:

$$\text{min}(Y) \setminus \text{min}(X) \iff X \geq Y \mid \text{true}.$$

In this rule, `min(X)` is removed, but only if there is a `min(Y)` with a smaller argument. So if a `min(X)` constraint is added, it should still be in the result, or else there has to be some constraint `min(Y)` in the result, with $X \geq Y$. Hence we can add the following early-fail rule:

```

observation(full), min(X) ==> check(min(X)).
result(min(Y)) \ check(min(X)) <=> X >= Y | true.
check(min(_)) <=> fail.

```

Since `min/1` has a functional dependency from \emptyset to its only argument (i.e. it is a singleton constraint), we can also write an early-fail rule for partial observations:

```

result(min(Y)), min(X) ==> X >= Y.

```

In general we say `c/n` is a *surviving* constraint if all of its removed occurrences satisfy the following property: either the rule also has a kept occurrence of `c/n`, or the rule body unconditionally (but possibly indirectly) re-inserts `c/n`.

Another example is the rule “`sum(X,A), sum(X,B) <=> C is A+B, sum(X,C)`”. In this case, we can check that once we add a `sum/2` constraint, the result must also contain a `sum/2` constraint with the same first argument:

```

observation(full), sum(X,A) ==> check(sum(X,A)).
result(sum(X,_)) \ check(sum(X,_)) <=> true.
check(sum(_,_)) <=> fail.

```

If we also know that the type of the second argument of `sum/2` is restricted to non-negative numbers, then we can infer that $C \geq A$ and $C \geq B$ and thus we can add this as a guard: “`result(sum(X,C)) \ check(sum(X,A)) <=> C >= A | true`”.

3.4 User-defined Early-fail Rules

The above kinds of early-fail rules can be added automatically by static program analysis. For a specific program, the programmer can also write additional user-defined early-fail rules. Those rules can take particular invariants into account that are hard to detect automatically, or they can be specific to the intended use of the program (possibly unsound if the program is used in unintended ways).

Normal (non-result-directed) program execution is not affected if the early-fail rules only apply when `observation/1` and `result/1` constraints are given. It is the responsibility of the programmer to write sound early-fail rules, i.e. no computations are pruned that could actually lead to the given result.

4 Experimental Evaluation

In order to evaluate the effect of adding early-fail rules, we consider the music generation program from the APOPCALEAPS system [12]. The benchmark consists of computing the probability of a simple piece of music of n measures of chords and drums. Still, the number of possible pieces is exponential in n .

Most of the output constraints of the APOPCALEAPS program are never-removed, so early-fail rules like the ones in Section 3.1 can be added. However, the constraint `beat/5`, which encodes rhythm, is not never-removed. The following probabilistic rule¹ removes `beat/5` constraints:

```

split_beat(V) ??
  meter(_,OD), phase(split_beats(M)), shortest_duration(V,SD),
  \ beat(V,M,N,X,D), next_beat(V,M,N,X,NM,NN,NX) <=> D<SD |
    D2 is D*2, X2 is X+1/(D2/OD),
    next_beat(V,M,N,X,M,N,X2), next_beat(V,M,N,X2,NM,NN,NX),
    beat(V,M,N,X,D2), beat(V,M,N,X2,D2).

```

¹ In case you are not familiar with CHRiSM syntax: a probabilistic rule is just a regular CHR rule that is optionally *not* applied; in this case, the probability of rule application depends on the stochastic experiment `split_beat(V)`.

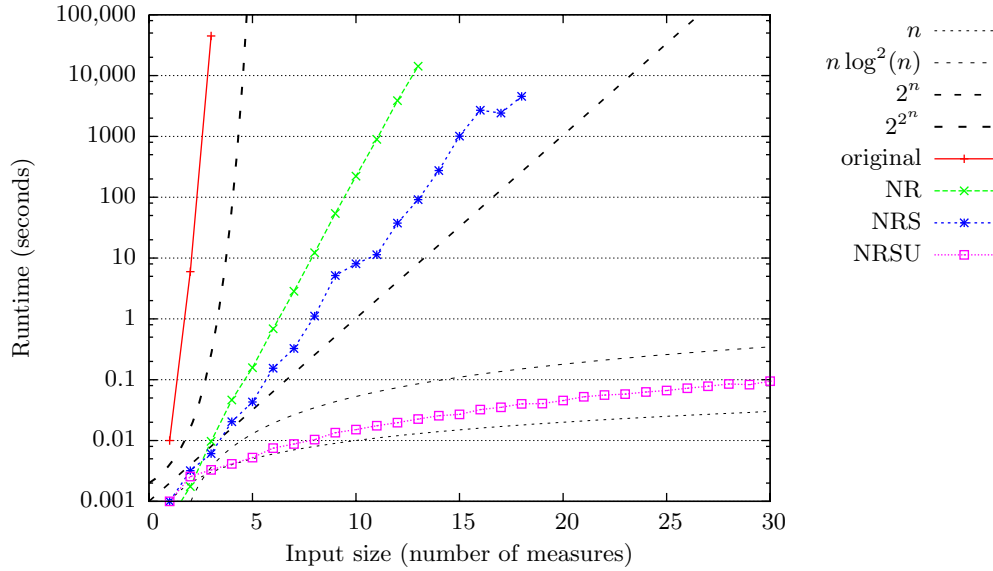


Fig. 1. Benchmark results.

This rule immediately reinserts a new `beat/5` constraint (even two), so `beat/5` is surviving (cf. Section 3.3). We can add the following early-fail rule: (the guard $Y \geq X$ can be added because the last argument of `beat/5` always increases)

```
observation(full), beat(A,B,C,D,N) ==> check(A,B,C,D,N).
result_beat(A,B,C,D,Y) \ check(A,B,C,D,X) <=> Y >= X | true.
check(_,_,_,_,_) <=> fail.
```

We can do better if we know more about the way the APOPCALEAPS program works. In particular, the beat-splitting rule splits beats of a given measure `M` when the program is in a phase denoted by `phase(split_beats(M))`. Once the next phase is entered, beats from previous measures become never-removed. This justifies the following user-defined early-fail rule:

```
observation(full), phase(split_beats(Ms)), beat(V,M,N,X,D)
==> M < Ms | check_NR_beat(V,M,N,X,D).
result_beat(V,M,N,X,D) \ check_NR_beat(V,M,N,X,D) <=> true.
check_NR_beat(V,M,N,X,D) <=> fail.
```

Figure 1 shows the benchmark results for several variants of the program, with increasingly sophisticated early-fail rules. All runtimes are averages over at least 5 random problem instances. The following variants are considered:

original: The original program, without any early-fail rules.

NR: The program with early-fail rules for all never-removed constraints

NRS: NR + an early-fail rule for the surviving constraint `beat/5`

NRSU: NRS + a user-defined early-fail rule

In this benchmark, the time complexity is reduced from exponential to almost linear. This is not surprising, since in this case there is only one explanation for every result, and if all never-fail rules are added, only a linear number of failing branches is investigated. Without never-fail rules, the entire tree is investigated, and it has an exponential number of leaves.

5 Conclusion

We have introduced an alternative execution mode for CHR, in which the aim is not to compute results for a goal, but instead to find derivations given both the goal and (part of) the result. The notion of surviving constraints (of which never-removed constraints are a special case) can be used to add early-fail rules which significantly improve the performance during result-directed execution.

Related Work. Pruning a search space by early detection of failure is obviously not a novel idea, and it has been applied in many contexts. A similar general idea underlies for example A* search [13], NOGOOD assertions in expert systems [14], search control for planning [15], and clause learning in SAT solvers [16]. Many of these existing approaches can undoubtedly be an inspiration in the context of result-directed CHR execution. However, there are important differences as well. For instance, clause learning is a dynamic approach and the result (a satisfying assignment) is not known in advance, while we derive early-fail rules statically and the result is given in advance.

Result-directed execution should not be confused with backward chaining as in Prolog [17] (and CHR^\vee [8]) or the goal-directed reasoning of ECLIPS [18]. In ECLIPS, goals are a control mechanism; they can be described in CHR as syntactic sugar for phase constraints implemented using the refined operational semantics.

In PRISM, efficient explanation search is greatly helped by tabling the probabilistic predicates [19]. In the context of CHR and CHRISM, tabling is problematic since rules implicitly take the entire execution state as input.

Future Work. An implementation that automatically detects never-removed and surviving constraints and adds the corresponding early-fail rules has still to be made.

The notions of never-removed and surviving constraints can be extended as in Section 4, to a conditional form: a constraint can be never-removed or surviving *under certain circumstances*. These circumstances can be properties of the current computation (as in Section 4) or properties of the goal and result; in any case, it is straightforward to come up with corresponding early-fail rules.

More complicated kinds of early-fail rules could be defined. In fact, every execution state property that behaves monotonically (e.g. a level mapping derived in a termination proof as in [20], even if they have no lower bound) could be converted to an early-fail rule. Also, a different approach for result-directed execution can be imagined, in which rules are applied in reverse, starting from the result and working towards the goal (perhaps guided by reverse early-fail rules).

For partial observations, the notion of early-*succeed* rules makes sense. That is, if we are in a state σ in which the partial result has been obtained, and we can somehow show that all further execution paths will succeed and not remove the partial result (e.g. if the partial result consists only of never-removed constraints and the program has no failing derivations), then we do not need to investigate the subtree rooted at σ .

The methods described here could be adapted to solve the related problem of determining if a derivation from G to R exists under a given (large) strategy class. For example, we may be interested in knowing whether there is a computation $G \mapsto^* R$ under the theoretical semantics ω_t . If the program is confluent, it suffices to execute it with goal G under any semantics that instantiates ω_t , and compare the result with R . However, if the program is non-confluent, that does not work. Instead one could make a general implementation of the theoretical semantics ω_t , which at each step computes all applicable transitions and chooses (in a backtrackable way) one transition to perform. Such a general implementation could generate all

possible computations starting from G , but if the program is highly non-confluent, the majority of those computations do not lead to R and can be pruned using early-fail rules (which should get priority over all original rules).

References

1. Frühwirth, T.: *Constraint Handling Rules*. Cambridge University Press (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming* **10**(1) (January 2010)
3. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: *ICLP '07*. Volume 4670 of LNCS, Porto, Portugal, Springer (September 2007) 224–239
4. Sneyers, J., Frühwirth, T.: Generalized CHR machines. [21] 143–157
5. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: *ICLP '04*. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 90–104
6. Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. In Hermenegildo, M., Niemelä, I., Schaub, T., eds.: *26th International Conference on Logic Programming*, Edinburgh, UK (July 2010)
7. Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems* **31** (2008)
8. Abdennadher, S., Schütz, H.: CHR^\vee , a flexible query language. In Andreassen, T., Christiansen, H., Larsen, H., eds.: *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*. Volume 1495 of LNAI, Roskilde, Denmark, Springer (1998) 1–14
9. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. In Schrijvers, T., Frühwirth, T., eds.: *CHR '06*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452, Venice, Italy (July 2006) 125–140
10. Sarna-Starosta, B., Schrijvers, T.: Transformation-based indexing techniques for Constraint Handling Rules. [21] 3–18
11. Duck, G.J., Schrijvers, T.: Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers, T., Frühwirth, T., eds.: *CHR '05*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, Sitges, Spain (2005) 109–124
12. Sneyers, J., De Schreye, D.: APOPCALEAPS: Automatic music generation with CHRiSM. In Downie, J., Veltkamp, R., eds.: *11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, Utrecht, The Netherlands (August 2010) Submitted.
13. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* **4**(2) (1968) 100–107
14. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* **9**(2) (1977) 135–196
15. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* **116**(1-2) (2000) 123 – 191
16. Zhang, H.: SATO: An efficient propositional prover. In: *14th International Conference on Automated Deduction*. Volume 1249 of LNCS, Springer (1997) 272–275
17. Clocksin, W.F., Mellish, C.S.: *Programming in Prolog*. Springer-Verlag (1984)
18. Homeier, P.V., Le, T.C.: ECLIPS: An extended CLIPS for backward chaining and goal-directed reasoning. In: *Proc. 2nd CLIPS Users Group Conference*. Volume 2 of NASA Conference Publication 10085, Houston, Texas, USA (1991) 213–225
19. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming* **8**(1) (2008) 81–109
20. Pillozzi, P., Schreye, D.D.: Automating termination proofs for CHR. In Hill, P.M., Warren, D.S., eds.: *25th International Conference on Logic Programming*. Volume 5649 of *Lecture Notes in Computer Science*, Pasadena, CA, USA, Springer (2009) 504–508
21. Schrijvers, T., Frühwirth, T., Raiser, F., eds.: *CHR '08: Proc. 5th Workshop on Constraint Handling Rules*, Hagenberg, Austria (July 2008)

Generic and Extensible Automatic Test Data Generation for Safety Critical Software with CHR

Ralf Gerlich

BSSE, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany,
ralf.gerlich@bsse.biz

Abstract. We present a new method for automatic test data generation (ATDG) applying to semantically annotated control-flow graphs (CFGs), covering both ATDG based on source code and assembly or virtual machine code. The method supports a generic set of test coverage criteria, including all structural coverage criteria currently in use in industrial software test for safety critical software.

Several known and new strategies are supported for avoiding infeasible paths, that is paths in the CFG for which no input exists leading to their execution. We describe the implementation of the method in CHR^v [1] and discuss difficulties and advantages of CHR in this context.

1 Introduction

Testing is one of the most important methods of analytical quality assurance of software-based systems, but also one of the most expensive, causing 50% of the effort for a typical software project [2] and nearly 80% for safety-critical software.

Software is categorised as safety-critical if its failure can lead to death or serious injury of humans, damage or loss of equipment or environmental harm. This kind of software is typically most complex as it has to handle and recover from many different types of failure, leading to individual components of hundreds of thousands or even millions lines of C or Ada code and thousands of interdependent functions.

Automatic Test-Data Generation (ATDG) aims to automate selection of test inputs and — if possible — the expected outputs. However, ATDG requires formal answers to two questions:

- Which criteria shall govern the selection of test data?
- How can we find samples that fulfill these criteria?

In practice the first question is typically answered by a list of well-known structural test criteria [3]. These criteria are defined based on the activation of specific portions of the control-flow graph (CFG) of the function under test. For example, the *all-nodes* criterion requires that for each node there is at least one test case by which the node is executed. Using the edges of the CFG, *all-edges* can be similarly defined.

Data-flow-based criteria are based on so-called definition-free paths. A path is *definition-free* regarding some variable v if no node in the path contains an assignment to v , except for the start and the end node. For example, the criterion *all-defs* requires that for each definition d of a variable v there must be at least one test case executing a definition-free path regarding v from d to a use u of v .

One approach to the second question is random testing [4], where inputs are selected randomly and the fulfillment of the criterion is checked afterwards [5]. Here statistical metrics on software quality can be derived. However, some portions of the CFG can only be reached for a small set of inputs and are therefore difficult to activate randomly.

Another approach is construction of some path in the CFG that fulfills the respective criterion, symbolically executing it to derive a set of equations and inequations and solving for the inputs [6]. However, in most cases there is a considerable set of so-called *infeasible paths* which cannot be activated by any input [7].

Gotlieb et al [8] propose a method for handling structured programs only consisting of **while**- and **if-else**-constructs. Multiple parts of an execution sequence can be processed in parallel, propagating information to be used for detecting and avoiding infeasible paths. Although a CFG can be emulated by a **while-if**-program, it is difficult to translate a CFG making proper use of the features of the method. Further, coverage goals have to be described by reference to the structured constructs, which specifically makes enforcing data-flow-based criteria tedious.

Godefroid et al [9] propose to randomly select an input and monitor the execution path for this input. Either the path matches the goal or there is a point in execution where a conflicting decision is made. The first conflicting decision is found and the path leading up to it is executed symbolically to derive a set of constraints. The constraints are then amended to enforce a decision matching the goal and the process is repeated with the solution of the constraint system as new input, if any. However it is possible that all paths with the selected prefix are either infeasible or do not match the goal.

We present an approach that overcomes these limitations and allows to compare different strategies, and discuss its implementation in CHR^V .

This paper is organised as follows: In Sect. 2 we briefly describe the relational semantics of CFGs, which is used in Sect. 3 to formalise some structural test criteria and to introduce solution rules and strategies. In Sect. 4 we discuss some of the advantages and disadvantages of CHR which became visible during implementation. Some results obtained from the prototype implementation are shown in Sect. 5, followed by our conclusions in Sect. 6.

2 Semantics of Control-Flow Graphs

For a detailed description of the theoretical foundation please refer to [7].

A CFG is a directed graph consisting of a finite set of nodes N and edges $E \subseteq N \times N$, with two special nodes, the entry node s and the exit node e . The entry node s has no incoming edges, while the exit node e has no outgoing edges. Further, every node n is reachable from s and e is reachable from every node n . We define E^+ to be the transitive closure of E . See Fig. 1(a) for an example CFG.

We extend a CFG to a program-flow graph (PFG) defining the semantics of the program. For this we first introduce the notion of memory state. A memory state represents the contents of memory relevant to the program at any given time during execution of a program. It can be represented, for example, as a tuple (v_1, \dots, v_n) , where v_i represents the value of a variable V_i in that state. In the following, we will use S to designate the set of memory states.

During execution, the program will proceed through its nodes, modifying the memory state by the statements inside the nodes. After execution of a node, a decision is required at which node the execution shall continue. A successor is eligible if and only if the predicate attached to the edge leading to the successor is fulfilled by the current memory state. Further, the modification of the memory state can be described as a relation between the memory state as found on entry to the node and the memory state as found on exit to the node.

In the following, the predicate attached to edge (u, v) is called $\mathcal{C}(u, v)$ and the relation for node u is called $\mathcal{B}(u)$. We assume that the predicates and the relations are decidable.

A graphical representation of a PFG is given in Fig. 1(b). Here we define S as the set of binary tuples of natural numbers \mathbb{N}^2 . In this example, the body of node 3 is to be understood as the relation $\mathcal{B}(3)$ with $(a_1, b_1) \mathcal{B}_3 (a_2, b_2) \Leftrightarrow b_2 = b_1 \wedge a_2 = a_1 - b_1$.

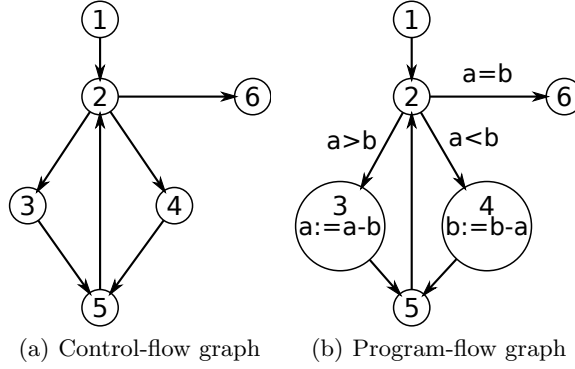


Fig. 1. Control- and Program-flow graphs for Euclid's algorithm

Given two nodes a and b we can define a relation $\mathcal{S}_{[a,b]} \subseteq S \times S$. For any two memory states $x, y \in S$, we have $x \mathcal{S}_{[a,b]} y$ if and only if input x to node a can be transformed to output y of node b along some execution path from a to b . We call $\mathcal{S}_{[a,b]}$ the *specification* of all paths from a to b .

Similarly, we can define a relation $\mathcal{I}_{[a,b]} \subseteq S \times S$ which relates the *outputs* of a to the *inputs* of b . We call $\mathcal{I}_{[a,b]}$ the *inner specification* of all paths from a to b , namely because it represents the transformations described by $\mathcal{S}_{[a,b]}$ minus the application of the bodies of a and b at the beginning respectively the end of the execution sequence.

The formalism can be used to model non-deterministic programs, but in practice non-deterministic programs are the exception.

3 Test-Data Generation

Using the relations defined in Sect. 2 we can now formalise several structural coverage criteria. For example, in order to cover some specific node $n \in N \setminus \{s, e\}$, we have to find some input $x \in S$ that will lead to execution of node n . This is only the case if there is some $y_1 \in S$ so that $x \mathcal{S}_{[s,n]} y_1$ holds. For deterministic PFGs, this necessary condition is also sufficient.

If we not only want to ensure the execution of n but also the completion of the program afterwards, we need $y_1, y_2, y_3 \in S$ so that $x \mathcal{S}_{[s,n]} y_1 \wedge y_1 \mathcal{I}_{[n,e]} y_2 \wedge y_2 \mathcal{B}(e) y_3$ hold.

Similarly, if we want to ensure execution of some edge (u, v) , we need some $y_1 \in S$ so that $x \mathcal{S}_{[s,u]} y_1 \wedge y_1 \in \mathcal{C}(u, v)$ holds. Definition-free paths can be formalised using appropriate extensions of $\mathcal{I}_{[a,b]}$ and $\mathcal{S}_{[a,b]}$ [7].

We can see that these conditions resemble the declarative content of a CHR^\vee goal. A constructive proof for satisfiability of such a goal will also yield candidate values for x and therefore the desired candidates for test inputs.

In this section we show a set of CHR^\vee rules, which can be used to produce such a constructive proof or to show non-satisfiability. We will use the built-in constraints from Tab. 1. Note that $\exists y : x \mathcal{S}_{[s,e]} y$ is satisfiable for some fixed x if and only if the program terminates on x . Therefore our CHR^\vee -program cannot terminate on all possible goals.

Table 1. Built-in constraints

Constraint	Semantics
<code>edge(U,V)</code>	$(u, v) \in E$
<code>reachable(U,V)</code>	$(u, v) \in E^+$
<code>body(U,X,Y)</code>	$x \mathcal{B}(u) y$
<code>cond(U,V,X)</code>	$x \in \mathcal{C}(u, v)$
<code>deffree(U,W,V)</code>	all paths from u to w are definition-free regarding v
<code>onallpaths(U,W,V)</code>	all paths from u to w proceed from u via v to w
<code>value(X,Var,Val)</code>	<code>Val</code> is the value of variable <code>Var</code> in state <code>X</code>

First of all, we can construct $x \mathcal{S}_{[a,b]} y$ using \mathcal{B} and \mathcal{I} , as already indicated in Sect. 2: For any given $x, y \in S$, $x \mathcal{S}_{[a,b]} y$ holds if and only if at least one of the following cases applies:

- $a = b \wedge x \mathcal{B}(a) y$.
- $\exists y_1, y_2 \in S : x \mathcal{B}(a) y_1 \wedge y_1 \mathcal{I}_{[a,b]} y_2 \wedge y_2 \mathcal{B}(b) y$.

This is implemented in Rule `spec_to_ispec` in Lst. 1, representing $x \mathcal{S}_{[a,b]} y$ as `spec(A,B,X,Y)` and $x \mathcal{I}_{[a,b]} y$ as `ispec(A,B,X,Y)`.

Now we can concentrate on solving $x \mathcal{I}_{[a,b]} z$, which holds if and only if at least one of the following cases applies:

- There is an edge from a to b and $x \in \mathcal{C}(a, b) \wedge x = z$ holds.
- The node n is a successor of a so that b can be reached from n and $\exists y \in S : x \in \mathcal{C}(a, n) \wedge x \mathcal{B}(n) y \wedge y \mathcal{I}_{[n,b]} z$ holds.

They are implemented in Rule `step_fwd`, providing a way of constructing a solution by iteratively stepping through the program in a forward direction. Analogously, Rule `step_bwd` implements iteratively stepping backwards through the program.

However, in some cases there are nodes which are traversed on every path from a to b . In the CFG shown in Fig. 1(a), for example, any path from node 3 to node 6 will traverse nodes 3, 5, 2 and 6. In case such a node n is known, we can split paths from a to b into two sub-paths from a to n and from n to b .

This split is implemented in Rule `split`. Note that for any pair of a and b there may be several different nodes n which are traversed on every path from a to b . However it can be shown that the choice of n is not relevant for the solution and a program consisting only of Rule `split` is confluent. In our prototype, we derive candidate split nodes using an efficient algorithm by Lengauer and Tarjan [10, 7].

The two subpaths introduced by `split` are not independent as the output of the first subpath is connected to the input of the second subpath via the body of n . However, we can exploit monotony and preservation properties of $\mathcal{I}_{[a,b]}$ to propagate information across these subpaths. One interesting property is the preservation of variable values. If we know that all paths from a to b are definition-free regarding some variable v , then we know that the value of v cannot change along any execution path from a to b . This notion is implemented in Rule `prop_var`.

Note that already a program consisting only of Rule `spec_to_ispec` and either Rule `step_fwd` or Rule `step_bwd` would implement the whole theory of \mathcal{I} and \mathcal{S} . The Rules `prop_var` and `split` are mainly required for improved search performance.

Therefore, in our actual implementation we allowed selectively disabling any of the latter rules and choosing at least one of the stepping rules. Such a configuration is called a *strategy*.

The `body/3` and `cond/3` built-in constraints are implemented as Prolog clauses, in turn using a custom-built solver for finite domain constraints. The latter is implemented in CHR as well, combining classic domain-filtering solution strategies with

```

spec_to_ispec @ spec(U,W,X,Z) <=>
  (U=W, body(U,X,Z));
  (body(U,X,Y1), ispec(Y1,U,W,Y2), body(W,Y2,Z)).
prop_var @ ispec(U,W,X,Y) ==> reachable(U,W), deffree(U,W,V) |
  value(X,V,V1), value(Y,V,V2), V1=V2.
split @ ispec(X,U,W,Z) <=> reachable(U,W), onallpaths(U,W,V) |
  ispec(X,U,V,Y), body(V,Y,Z), ispec(Y,V,W,Z).
step_fwd @ ispec(X,U,W,Z) <=>
  (edge(U,W), X=Z, cond(U,W,X));
  (edge(U,V), reachable(V,W),
   cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
step_bwd @ ispec(X,U,W,Z) <=>
  (edge(U,W), X=Z, cond(U,W,X));
  (edge(V,W), reachable(U,V),
   ispec(X,U,V,Z), body(V,Z,Y), cond(V,W,Z)).

```

Listing 1: CHR^V-Implementation

axiomatic rules. This way inconsistencies such as $a < b, b < a$ can be detected more efficiently using the transitivity and irreflexivity of $<$ than by pure domain filtering.

For testing deterministic selection of test inputs is not desirable. Therefore we implemented a probabilistic version of the program, in which in every step first the Rules `spec_to_ispec`, `split` and `prop_var` are applied exhaustively, if enabled. After that, stepping forward or backward is selected with probability p and completion of a subpath by direct edge traversal is selected with probability $1 - p$, if applicable. If both `step_fwd` and `step_bwd` are enabled — called a *mixed* configuration — they are applied with the same probability. The construction of a path now becomes a Bernoulli experiment, favouring shorter paths, but this bias can be at least partially compensated by varying p .

If stepping through a further node is selected, one of the alternative successors is selected according to a slightly skewed uniform distribution. Here nodes inside a loop are favoured over nodes by which the loop is exited in order to avoid constant and minimal mean iteration counts for inner loops [7].

4 Implementation Issues

Although the theoretical construction theorems are quite similar to the declarative semantics of CHR rules, it was not possible to transform them directly.

For example, the desired probabilistic behaviour as described in Sect. 3 can be modelled with Probabilistic CHR (PCHR) [11], but only by splitting up alternatives of Rules `step_fwd` and `step_bwd` into individual rules and thereby giving up the connection to the declarative semantics of CHR^V. Additionally, the second alternative of these rules has to be boxed into another constraint to delay the selection of the successor respectively predecessor node. Otherwise, each of the possible intermediate nodes would weigh in as an alternative to closing a subpath, making the probability of stepping dependent on the number of available intermediate nodes.

From a first look it seems that CHRiSM [12] provides a better integration of PCHR with CHR^V. Unfortunately, it was not available during implementation of the prototype.

Also, the first alternatives of Rules `step_fwd` and `step_bwd` are actually only present in the first of two cases of the stepping theorem, namely the case $(u, w) \in E$. So in these rules, `edge(U,W)` actually is a guard, but as CHR^V does not allow individual guards for alternatives, the constraint had to be moved into the body

of the first alternative. Operationally, this does not make a difference as the first alternative does fail if there is no edge from u to v , just as if it had not been activated due to the guard.

Further, the second alternatives of Rule `step_fwd` and `step_bwd` actually correspond to a comprehensive union over all cases of $(u, v) \in E, (v, w) \in E^+$. As `edge/2` is a built-in constraint, search over its solutions is not supported by the declarative semantics of CHR^\forall . The program again is only operationally correct and only because the underlying host, SWI Prolog, allows search on `edge/2`, respectively our search extension for PCHR considers all applicable instances of a rule.

Still, these constraint solvers would be hardly manageable without CHR at all. At 26 constraints in 126 rules for the built-in solver and 45 constraints — many of them part of the probabilistic selection implementation or used for debugging purposes — in 74 rules for the path construction solver, a manual implementation is not feasible.

A CHR rule expresses interdependencies of many constraints. These interdependencies are difficult to handle with the classical “separation-of-concerns” approach to limiting complexity. Therefore a CHR compiler taking the burden of keeping an implementation of that size consistent is a huge relief for the developer.

5 Evaluation

We have applied the program to several example programs and determined the strategies performing best and worst as shown in Tab. 2. During the experiments, the length of the constructed path and the time required for construction were recorded. The best strategy was determined based on a fit of a second-order polynomial to the data as well as the observed scatter. The best overall strategy is marked with a †-symbol.

Table 2. Comparison of Strategies

Program	Goal	best		worst
		without <code>split</code>	with <code>split</code>	
Fibonacci	feasible path	<code>step_bwd</code> †	<code>step_bwd</code>	<code>mixed+split</code>
Selection Sort	feasible path	<code>step_fwd</code> †	n/a	<code>step_fwd+split</code>
<code>strcmp</code> without <code>break</code>	result = 0	<code>step_bwd</code> †	n/a	<code>mixed+split</code>
<code>strcmp</code> with <code>break</code>	result = 0	mixed	<code>step_bwd</code> †	<code>mixed+split</code>
Insert into array	cover node	mixed	<code>step_bwd</code> †	<code>step_fwd+split</code>

Several of the strategies showed a notable scatter in runtime due to backtracking. In some cases, the runtime impact of backtracking is so high that only very short paths can be constructed in an acceptable time, as shown in Fig. 2(b).

6 Conclusions and Outlook

We have presented a novel approach to ATDG and its implementation in CHR^\forall . The approach supports several combinable path construction strategies, none of which is optimal, and a generic set of structural coverage criteria, including all industrial criteria for safety-critical software.

Unfortunately, none of the currently available CHR compilers is formally qualified according to industry standards. Such a qualification is necessary for acceptance of development tools in the context of safety-critical applications. In contrast, the

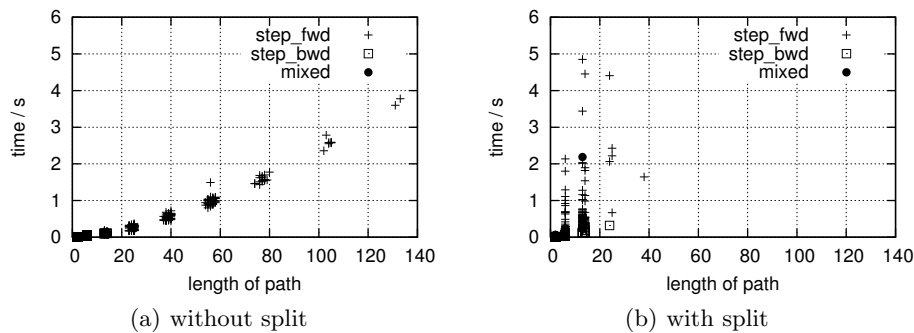


Fig. 2. Performance measurements for Selection Sort

argument in favour of CHR-based tools is strongly supported by the close connection of theory and practice in CHR and most of its variants.

Further research will focus on the possible application of CHRiSM as well as on integration with results from static program analysis such as abstract interpretation. Also, several projects for application on real-life safety-critical software are already defined and a toolchain for the language C based on the method is currently under development.

References

1. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer Verlag (2003)
2. Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1995)
3. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE '82: Proceedings of the 6th international conference on Software engineering, IEEE Computer Society Press (1982) 272–278
4. Hamlet, R.: Random testing. In Marciniak, J., ed.: Encyclopedia of Software Engineering. Wiley (1994) 970–978
5. Gerlich, R., Gerlich, R., Boll, T.: Random testing: From the classical approach to a global view and full test automation. In: RT '07: Proceedings of the 2nd international workshop on Random testing, ACM (2007) 30–37
6. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE), IEEE (2004) 25–34
7. Gerlich, R.: Verallgemeinertes Rahmenwerk zur constraintbasierten Testdatenerzeugung aus Programmflussgraphen. PhD thesis, Faculty of Engineering and Computer Science, University of Ulm, Germany (2009)
8. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. SIGSOFT Softw. Eng. Notes **23**(2) (1998) 53–62
9. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM (2005) 213–223
10. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**(1) (1979) 121–141
11. Frühwirth, T., Pierro, A.D., Wiklicky, H.: Probabilistic Constraint Handling Rules. In: WFLP 2002, 11th International Workshop on Functional and (Constraint) Logic Programming. Volume 76 of Electronic Notes in Theoretical Computer Science., Elsevier (November 2002) 1–16
12. Sneyers, J., Meert, W., Vennekens, J.: CHRiSM: CHance Rules induce Statistical Models. In: Proceedings of the Sixth International Workshop on Constraint Handling Rules (CHR'09). (July 2009) 62–76

MTSeq: Multi-touch-enabled CHR-based Music Generation and Manipulation

Florian Geiselhart¹, Frank Raiser¹, Jon Sneyers², and Thom Frühwirth¹

¹ Faculty of Engineering and Computer Science, Ulm University, Germany
`firstname.lastname@uni-ulm.de`

² Department of Computer Science, K.U.Leuven, Belgium
`firstname.lastname@cs.kuleuven.be`

Abstract. We present MTSeq, an application that combines GUI-driven multi-touch input technology with the CHR-based music generation system AOPCALEAPS and an advanced audio engine. This combination leads to an extended user experience and an intuitive, playful access to the CHR music generation system, and thus introduces CHR to musicians and other non-computer-scientists in an appropriate way. The application is fully modularized and its parts are loosely interconnected through a standard IP networking layer, so it is optionally distributable across multiple machines.

1 Introduction and Goals

In our application, we show how the Constraint Handling Rules-based (CHR [3]) music generator *AOPCALEAPS*[8] is driven via a loose-coupled, modern, and multi-touch-enabled GUI, which communicates with the CHR backend. An extended audio engine allows high-quality playback and real-time manipulation of the generated music. In combination, these components form a highly interactive music generation and manipulation environment called *MTSeq* (as an acronym of *Multi-Touch* and *Sequencer*).

The most important goal of the MTSeq application is to make the AOPCALEAPS CHR application accessible to non-computer-scientists and musicians. This creates interest in CHR in user groups who did not consider CHR as an application language by now.

Another goal is to make use of new and innovative multi-touch technology for usability improvements and a modern GUI. This technology is especially interesting if an application demands lots of parallel manipulation actions, like it is common in musical environments, e.g., for parallel effects parameter manipulation. Thus, AOPCALEAPS is a suitable candidate for use with a multi-touch interface.

Given those preconditions, a GUI design goal is to resemble the look-and-feel of common musical controllers, which are widely used among musicians. These controllers heavily depend on hardware controllers, like rotary controller, sliders, and buttons, which are good for quick and intuitive manipulation especially in live situations. They are already familiar as a control paradigm to the targeted user group on the one hand, but on the other hand, the multi-touch hardware furthermore promotes the realization of such GUI widgets.

Our paper first gives an overview of the underlying CHR music generation system *AOPCALEAPS* in Section 2. Afterwards the GUI and audio extensions that were made are described in detail in Section 3. In Section 4, we sum up the experiences gained and lessons learned throughout the implementation.

In addition to this paper, a short demonstrational video is available³.

³ <http://www.uni-ulm.de/in/pm/forschung/themen/chr/info/downloads.html>

2 AOPCALEAPS

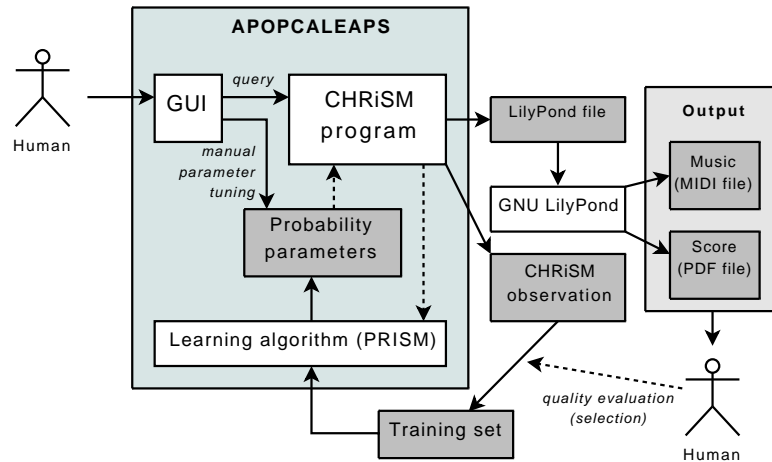


Fig. 1. An overview of the AOPCALEAPS system [8].

The core generation system of our application is formed by AOPCALEAPS[8] (an acronym for “Automatic POP Composer And LEArner of ParameterS”, a music generation application which is built upon CHR[3], the probabilistic logic language PRISM[6] and its corresponding extension to CHR, called CHRiSM[9]). The remainder of this section is based on [8] and provides an overview of AOPCALEAPS.

Figure 1 gives a schematic overview of AOPCALEAPS. In the original AOPCALEAPS version, a minimal graphical user interface provides a front-end to the underlying CHRiSM program. This interface essentially allows the user to tweak an input query for the CHRiSM program, specifying some desired properties of the generated music. The default query is as follows:

```

voice(melody), shortest_duration(melody,16),
voice(bass), shortest_duration(bass,8),
voice(chords), shortest_duration(chords,8),
voice(drums), shortest_duration(drums,16),
instrument(melody,'soprano sax'),
instrument(bass,'electric bass (pick)'),
instrument(chords,'electric guitar (jazz)'),
set_range(melody,c,4,-5,16), max_jump(melody,5),
set_range(bass,c,3,-17,5), max_jump(bass,17),
chord_style(offbeat), max_repeat(melody,2),
key(major), meter(2,4), tempo(120), measures(8)
  
```

The above query indicates that we want a piece with four voices: melody, bass, chords and drums. The shortest possible note for the melody and drums is set to a 16th note, while for the bass and chords it is set to an 8th note. Names of MIDI instruments to be used to render the voices are given. The range of the melody is set to the interval of 5 semitones below central C to 16 semitones above central C. The biggest interval between two consecutive melody notes is set to 5 semitones. The bass has a lower range and is allowed to make bigger jumps. Chords are preferably on off-beats, and the melody should not have more than two consecutive repeated notes. The piece should be in a major key. The meter is 2/4, the tempo is 120 bpm, and the length of the piece to be generated is 8 measures.

Based on the query and probability parameters inside the CHRiSM program, CHRiSM generates output, rendered by LilyPond [5] as both a score and a MIDI file.

Additionally, the original APOPCALEAPS system in principle supports an iterative and interactive learning process, where users listen to the generated music and select the good pieces according to their own taste. The selected pieces are used as a training set for a learning algorithm that adjusts the probability parameters. The idea is that this iterative interactive process leads to a personalized music generation system.

However, due to CHRiSM not being able to deal efficiently with large output spaces yet, the learning features are computationally too expensive to be tested in practice on non-trivial examples. So for now, the CHRiSM program is driven by manual-tuned probability parameters.

The core component of the APOPCALEAPS system is, as stated above, a CHRiSM program. It consists of about 50 CHRiSM rules (about 150 lines of code). Besides the actual program, there is some auxiliary code (about 100 lines of code) and the code to write out the output in LilyPond syntax (about 150 lines of code).

The program uses 7 parametrized probabilistic experiments, which give rise to 92 probability distributions in total.

3 Multi-touch GUI and Audio Rendering

To extend the APOPCALEAPS system according to our goals, there were mainly two work areas. The first is the development of a multi-touch-enabled GUI in a suitable programming language, the second is the development of an extended audio processing system.

3.1 GUI Basics and CHR Interfacing

For the sake of rapid multi-touch UI prototyping on a limited time budget, Adobe Flash⁴ was chosen as a base for GUI development. It allows a rapid and easy way of GUI widget design via vector based drawing and is suitable for multi-touch-enabled GUIs because of its programming model. Thus it is being often used in multi-touch applications (e.g., [2, 7, 10] and numerous non-academic projects⁵).

The multi-touch tracking data used for control is encapsulated in Open Sound Control [1] (OSC) UDP messages following the TUIO [4] standard, and made available via a small UDP-to-TCP gateway program that Flash can connect to. We decided to re-use this existing way to communicate with the other components, as the audio processing environment is already compatible to the OSC standard, and the network-based architecture allows us to run APOPCALEAPS within a Linux virtual machine, while the GUI is running on a Windows host.

The main task to be done beneath the GUI was to develop a proxy-like program with the following features:

- Send and receive OSC messages via UDP/IP networking
- Keep a state model of all APOPCALEAPS parameters which is modifiable through OSC
- Serialize its state to a textual goal file as an input to APOPCALEAPS
- Start and control the generation and signal its end through OSC to the GUI

⁴ <http://www.adobe.com/products/flash/>

⁵ <http://www.nuigroup.com>

The first and second features are handled by a small Java program and existing OSC libraries. On generation, it serializes the parameter variables to a text file in the appropriate format derived from the structure of the goal handler of APOP-CALEAPS. To control the generation process, an already existing set of shell scripts from the first minimal GUI is reused. These scripts are called directly from the Java program, so it is always in control of the generation process.

This combination of components makes it possible to modify the parameters in real-time when the GUI is changed. The actual generation is ran on demand, in a synchronous way - that means the playback is stopped when the user triggers generation, and he has to wait until the process finishes. This is primarily necessary because the generation process is not done in real-time, but also because the audio processing environment needs about 5 seconds to unload the old MIDI file and to reload the new one.

3.2 GUI structure



Fig. 2. GUI Design and Sections

The basic sections of the GUI are pointed out in Figure 2. In the upper part of the GUI, section (1) represents the generator controls. The parameter values are sent to the generator proxy in real-time, but as described above, they do not come into effect until the generation is triggered by the GO button on the right.

The lower part of the GUI allows real-time manipulation of playback and effects. It consists of a tabbed group of effect controls (2), which can be wired to the X-Y-controller pad on the right (3) in many ways. The controller pad therefore exposes the number of fingers, the x and y value of fingers and the distance and angle between certain fingers as a numeric controller value. In the very bottom part of the GUI, the global transport controls can be found. They control basic playback parameters like speed, play/stop, and volume.

Interaction Design The whole interface and interaction is designed in style of a electronic musical device to allow quick access for unexperienced users. Especially the software rotary controller in our GUI follows this paradigm through a 2-finger control gesture, similar to the way a hardware rotary controller, e.g. on a mixing desk, might be used in the real world. In addition, the X-Y-controller resembles a real-world class of electronic musical devices, for example known as *KORG Kaoss Pad*⁶. However, an advantage of our solution is multi-touch support, which allows more parameters to be controlled at one time.

3.3 Advanced Audio and Effect Rendering

To improve the audio quality and to enable the use of effects, we integrated the commercial audio processing application *Bidule*⁷. Bidule provides a graph-oriented way of building audio processing chains and playing back audio data. Almost all of its parameters are remote controllable via OSC. We designed a so-called *patch*, that features a MIDI player and MIDI processing, a third-party software-based instrument for playing back the MIDI data from APOPCALEAPS in high quality, and a configurable set of effects that can be applied to the wave data before outputting it to the sound card. These measures greatly improve the overall user experience, especially when compared to the quality and possibilities of direct MIDI playback through a regular computer on-board sound card.

4 Conclusion

With the MTSeq system, we created an appealing application which helps to attract end users to APOPCALEAPS and thus, to CHR. We proposed and used a new and simplistic approach to the communication with CHR through a special proxy-like Java application, because this method allowed us to reuse wide parts of the APOPCALEAPS handling and control mechanisms with minor changes.

Using Adobe Flash, we were able to rapidly create a rich graphical UI, but with the consequence of having to communicate via a third party application (FLOSC⁸) that helps bypassing the Flash sandbox, because Flash normally is not able to create a listening network socket, which would be required for OSC communication.

Another problem we encountered resulted from the development process itself. As we only were able to access real multi-touch hardware from time to time, the main work had to be done via a TUIO simulator⁹ to emulate a real multi-touch table. But the behaviour differences of real hardware compared to the simulator were significant, so major parts of the interaction concept had to be altered to maintain functionality on a real hardware table. This especially changed the way the wiring area works, and introduced a de-bouncing routine for switches and buttons.

Besides this, the application still leaves room for improvements. With an extension of APOPCALEAPS as a pseudo-real-time MIDI generation engine, the currently blocking-mode generation part of our application could work in real-time. This would enable the audio engine to play back a real-time MIDI data stream coming from APOPCALEAPS instead of generated MIDI files. The stream itself might be altered directly through the GUI parameters, which would result in a more responsive user experience and less waiting time compared to our current approach. However, this tends to lead to a complete rework of our CHR proxy application, as the basic concept doesn't support real-time parameter modification very well.

⁶ <http://korg.com/product.aspx?pd=269>

⁷ <http://www.plogue.com>

⁸ See <http://www.benchun.net/flosc/> and <http://code.google.com/p/flosc/>

⁹ A part of the Reactivision framework (<http://reactivision.sourceforge.net/>)

This might, in turn, give rise to the (currently untested) direct Java interfacing of CHRiSM and PRISM through the external language interface of the underlying *B-Prolog* system.

Another feature that we did not implement to keep our GUI as simple as possible are controls for the probability parameters inside APOPCALEAPS. This might be worked out in a future version of MTSeq, but as the modification requires recompilation of parts of APOPCALEAPS to come into effect, this may introduce an additional waiting phase during music generation. Furthermore, a different GUI extension could be a direct and graphical manipulation of CHR rules and parameters on a score, as they represent the relations between the notes.

A last possible extension point is the extension and generalization of the OSC protocol subset between flash and APOPCALEAPS/CHR, as the current version only features the most basic parameters for communication and control of the APOPCALEAPS system. There are for example no error messages provided to the GUI if something goes wrong during the generation process.

References

1. OpenSound Control: state of the art 2003. National University of Singapore, Singapore, Singapore (2003)
2. Chen, C.H., Nien, K.H., Wu, F.G.: Design a multi-touch table and apply to interior furniture allocation. In: Universal Access in Human-Computer Interaction. Intelligent and Ubiquitous Interaction Environments. Lecture Notes in Computer Science, vol. 5615, pp. 13–19. Springer Berlin / Heidelberg (2009), <http://www.springerlink.com/content/4h97170251842644/>
3. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
4. Kaltenbrunner, M., Bovermann, T., Bencina, R., Costanza, E.: Tuio - a protocol for table based tangible user interfaces. In: Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation (GW 2005). Vannes, France (2005)
5. Nienhuys, H.W., Nieuwenhuizen, J.: LilyPond, a system for automated music engraving. In: Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003). Firenze, Italy (May 2003)
6. Sato, T.: A glimpse of symbolic-statistical modeling by prism. Journal of Intelligent Information Systems 31, 161–176 (2008)
7. Simona Vlad, R.V.C., Nicu, A.I.: Optical multi-touch system for patient monitoring and medical data analysis. In: International Conference on Advancements of Medicine and Health Care through Technology. pp. 279–282. Springer-Verlag, Berlin, Heidelberg (2009)
8. Sneyers, J., De Schreye, D.: APOPCALEAPS: Automatic music generation with CHRiSM. In: Downie, J., Veltkamp, R. (eds.) 11th International Society for Music Information Retrieval Conference (ISMIR 2010). Utrecht, The Netherlands (August 2010), submitted
9. Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. In: Hermenegildo, M., Niemelä, I., Schaub, T. (eds.) 26th International Conference on Logic Programming. Edinburgh, UK (July 2010)
10. Strijkers, R., Muller, L., Cristea, M., Belleman, R., de Laat, C., Sloot, P., Meijer, R.: Interactive control over a programmable computer network using a multi-touch surface. In: Computational Science - ICCS 2009. Lecture Notes in Computer Science, vol. 5545, pp. 719–728. Springer Berlin / Heidelberg (2009), <http://www.springerlink.com/content/p2r7672p383gh124/>