A compilation schema from Constraint Handling Rules into Action Rules

Tom Schrijvers *Neng-Fa Zhou* *Bart Demoen*

Report CW 449, June 15, 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A compilation schema from Constraint Handling Rules into Action Rules

Tom Schrijvers *Neng-Fa Zhou* *Bart Demoen*

Report CW 449, June 15, 2006

Department of Computer Science, K.U.Leuven

Abstract

CHR is a popular high-level language for implementing constraint solvers and other general purpose applications. It has a well-established operational semantics and quite a number of different implementations, prominently in Prolog. However, there is still much room for exploring the compilation of CHR to Prolog. Nearly all implementations rely on attributed variables. In this paper, we explore a different implementation target for CHR: B-Prolog's Action Rules (ARs). As a rule-based language, it is a good match for particular aspects of CHR. However, the strict adherence to CHR's refined operational semantics poses some difficulty. We report on our work in progress: a novel compilation schema, required changes to the AR language and the preliminary benchmarks and experiences.

Keywords : Constraint Handling Rules, Action Rules, B-Prolog.
CR Subject Classification : D.3.2, D.3.3, D.3.4

A Compilation Schema from Constraint Handling Rules into Action Rules

Tom Schrijvers[2⋆], Neng-Fa Zhou[1] and Bart Demoen[2]

[1] Department of Computer Science
K.U.Leuven, Belgium
{toms,bmd}@cs.kuleuven.be
[2] Department of Computer & Information Science
CUNY Brookly College & Graduate Center
zhou@csi.brooklyn.cuny.edu

Abstract. CHR is a popular high-level language for implementing constraint solvers and other general purpose applications. It has a well-established operational semantics and quite a number of different implementations, prominently in Prolog. However, there is still much room for exploring the compilation of CHR to Prolog. Nearly all implementations rely on attributed variables. In this paper, we explore a different implementation target for CHR: B-Prolog's Action Rules (ARs). As a rule-based language, it is a good match for particular aspects of CHR. However, the strict adherence to CHR's refined operational semantics poses some difficulty. We report on our work in progress: a novel compilation schema, required changes to the AR language and the preliminary benchmarks and experiences.

1 Introduction

Constraint Handling Rules (CHR) [13] is a rule-based programming language commonly embedded in a host language. It is a powerful yet relatively simple programming language that combines elements of Constraint (Logic) Programming (CLP) and rule-based languages. CHR's is intended as a language for implementing user-defined application-tailored constraint solvers, but it is also used as a general programming language.

Several implementations of CHR exist and most are embedded in Prolog [16, 18, 21] and HAL [10, 17]. CHR implementations for Java [1, 24] and Haskell [22] exist as well. The implementation [16] in SICStus Prolog is generally considered the reference implementation because it is historically the first full-fledged CHR system. It implements an efficient compilation schema in terms of attributed variables. More recently, the formulation of the refined operational semantics of CHR [11] has captured the essentials of the reference implementation on a more formal level.

Applications of CHR in the domain of constraint solving, scheduling and optimization, are university lecture time tabling [2] and optimal placement of

⋆ Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

local telecommunication transmitter stations [14]. However, the customizability of CHR is really advantegous in less traditional constraint solving domains. Natural language processing is one such domain, with for example logical grammars based on CHR [8]. Type inference, type checking [9] and type system design [22] is a another domain CHR is used in. One more example of a CHR application domain is Multi-Agent Systems: reasoning about hypotheses [4], semantics of agent communication languages [5] and modeling the environment of an agent [23] are just a few of the problems solved with CHR.

AR (Action Rules) is a rule-based event-handling language originally designed for programming constraint propagation [25]. It has been employed in the implementations of several highly efficient constraint solvers. There are two major differences between AR and CHR: (1) CHR allows multi-headed rules while AR only accepts single-headed rules; and (2) unlike in CHR where primitive delay conditions are hidden in heads and guards of rules, all primitive delay conditions must be explicit in AR. It is not obvious how CHR can be compiled into AR, a seemingly lower-level language.

The goal of this paper is to explore a novel Prolog implementation of CHR in terms of Action Rules. Traditionally, CHR is compiled to Prolog using attributed variables [15], a low-level primitive for implementing constraint solvers. We argue that because of its rule-based nature, higher level of expressivity, Action Rules may provide a fair alternative compilation target. In our exploration we hope to assess this claim and gain new insights into the efficient compilation of CHR.

The rest of this paper is structured as follows. In the next section, we provide a brief overview of CHR-related concepts that are important for this paper. Subsequently, in Section 3 we introduce the Action Rules language. The novel compilation schema from CHR to AR is presented in four steps. First, in Section 4, as basic compilation schema is presented for two-headed rules. Second and third, Sections 5 and 6 adapt this schema respectively for single-headed and multi-headed rules. Fourth, a number of optimizations to this basic schema are sketched in Section 7. We report on a preliminary experimental evaluation in Section 8. Finally, in Section 9 we summarize and discuss our experiences.

2 Constraint Handling Rules

We assume the reader to be already familiar with the general aspects of the CHR language, its syntax and operational semantics.

The compilation schema we will present, implements the refined operational semantics [11], which is the de facto standard operational semantics of CHR implemented by all major CHR systems. An important notation in the refined semantics, is the *occurrence* of a constraint symbol and the order of occurrences.

Example 1. An example should clarify that - we use `*=>` when it doesn't matter whether the symbol is `==>` or `<=>`:

```
p(A)1, q          *=> ...
p(B)2, p(C)3      *=> ...
p(D)4 \ q         <=> ...
q \ p(E)5         <=> ...
p(F7) \ p(G)6     <=> ...
```

The constraint symbol `p/1` has 7 occurrences in the above code (since `q` differs from any `p` literal). In this example the subscript of the occurrences denotes the order of the occurrences: this order is used by the refined semantics to execute the actions when a constraint is posted to the store.

In general the order is textual following the rules, and in a head it is from left to right, except for the simpagation rule, where an occurrence of `p/1` to the right of the \ comes before the occurrence of `p/1` to the left.

The refined semantics follows a depth first execution strategy. One constraint is activated at a time and for this constraint the occurrences are visited in order. At each occurrence, the corresponding rule is attempted. If the rule succeeds, execution of the active constraint is temporarily suspended, in favor of depth first execution of the rule body. For further details we refer to [11].

3 AR: Action Rules

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [25]. It has been used to implement the efficient finite-domain and finite-set domain solvers in B-Prolog.

An action rule takes the following form: "$H, G, \{E\}$=>B" where H (called the *head*) is an atomic formula that represents a pattern for agents, G (called the *guard*) is a conjunction of conditions on the agents, E (called *event patterns*) is a non-empty disjunction of patterns for events that can activate the agents, and B (called *action*) is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if E together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

For this paper, we are interested in the following event patterns:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this event pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `event`(X, T): the general form of user-defined events, where X is a variable, called a *channel*, and T is a Prolog term, called an *event object*.
- `event`(X): Same as `event`(X, T) but no event object is used.

In general, an action rule may specify several event patterns such as `ins` patterns on several variables. However, co-existing patterns of `event`/2 must all have the

same variable as the second argument so the variable always references the event object when the rule is triggered by an event of any pattern. Event patterns of `event`/2 are allowed to co-exist with `ins` patterns as well. Nevertheless, for each co-existing `ins`(X) pattern, there must be a condition `var`(X) in the guard so that the action is never executed when the rule is triggered by an `ins` event.[3]

For an agent, a rule is *applicable* to it if the agent matches the head of the rule and the guard is satisfied. When an agent is created, the system checks if the action rule in its predicate is applicable to it.[4] If so, the agent will be suspended until it is *activated* by one of the events specified in the rule. Whenever the agent is activated by an event, the condition of the action rule is tested *again*. If it is met, the action is executed. The agent does not vanish after the action is executed, but instead sleeps until it is activated again. There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it.

Events are posted though a channel to agents by the system or by the user program. The built-in primitive `post_event`(C, T) posts a general event to the agents connected to C, where C is a *channel*. The activated agents are first connected to the active chain of agents and are then executed one at a time. Therefore, agents are activated in a *breadth-first* fasion. The second argument can be omitted if no information needs to be transmitted to the activated agents.

There is no primitive for killing agents explicitly. As described above, an agent never disappears as long as action rules are applied to it. An agent vanishes only when a commitment rule is applied to it. Consider the following example.

```
p(X,Flag), var(Flag), {event(X,O)} =>
        write(O),Flag=1.
p(X,Flag) => true.
```

An agent defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. So, when a second event is posted, the action rule is no longer applicable and thus the commitment rule after it will be selected.

One question arises here: what happens if there will never be another event on `X`? In this case, the agent will stay forever. If we want to kill the agent immediately after it is activated once, we have to define it as follows:

```
p(X,Flag), var(Flag), {event(X,O),ins(Flag)} =>
        write(O),Flag=1.
p(X,Flag) => true.
```

In this way, the agent will be activated again after `Flag` is bound to 1, and be killed after the failure of the test `var(Flag)`.

[3] It is the programmer's responsibility to ensure that the action is not executed if the rule is triggered by an `ins` event.

[4] Since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in the guard, the agent will remain the same after an applicable rule is found.

4 The Basic Translation Schema

This section gives the general schema for translating CHR rules into AR rules. We focus on translating double-headed CHR rules. For single headed rules, a head constraint is treated as the partner of itself (discussed in Section 5), and rules with more than two constraint patterns in the head are translated to double-headed rule (discussed in Section 6) before translating them into AR rules.

4.1 Preparation

Without loss of generality we assume that all rules in the given CHR program have been transformed into the *head normal form* [17] where all matching operations are encoded explicitly in the guards of the rules. Hence, all the arguments of the constraint patterns in the program are unique variables.

For each constraint $p(\overline{X})$, (where $\bar{X}$ stands for $X_1, \ldots, X_n$), an event of the following form is posted when the constraint is added into the store and when any variable in it is instantiated:

$$\texttt{constr}(Cno, Alive, \overline{X})$$

where *Cno* is a unique identifier of the constraint, and *Alive* is a logic variable, called a *status variable*, used to indicate aliveness of the constraint (the variable is free if the constraint is alive and is bound to 0 if it has been removed from the store). We need the following primitives:

- `gen_constr_id`(*Cno*): Generates a new identifier *Cno*. The nature of the identifier is immaterial; an integer is used in our implementation.
- `get_store/`(*Store*): Returns the constraint store *Store* which holds all posted constraints; the constraint store is a variable which acts as a channel for posting events,
- `not_in_history`($CNo, PartnerNo, RuleNo$):
- `add_to_history`($CNo, PartnerNo, RuleNo$): The primitive `not_in_history` succeeds if the tuple $(CNo, PartnerNo, RuleNo)$ is not included in the globally accessible propagation history, and the primitive `add_to_history` adds the tuple $(CNo, PartnerNo, RuleNo)$ into the propagation history. The data structure for the history is immaterial at this moment: it could very well be an open-ended list.

For each occurrence of a constraint symbol in the heads, we use a unique identifier, called an *occurrence identifier*, to denote it, which is composed of the constraint symbol, the arity, and the number of the occurrence. For example, `p(D)` in Example 1 has the identifier `p_1_4`, which means that `p(D)` is the 4th occurrence of `p/1` in the program.

For each constraint symbol, we define two lists of occurrence identifiers: The *primal occurrence* list contains the occurrence identifiers of that symbol and the *partner occurrence* list contains the identifiers of the partner occurrences. The two lists overlap if there is a rule with two identical constraint symbols in the

head. The ordering of primal occurrences is not important, but the occurrence identifiers in the partner list must be in the order specified by the refined operational semantics. For example, the partner occurrence list for `p/1` in Example 1 is `[q_0_1, p_1_2, p_1_3, q_0_2, q_0_3, p_1_6, p_1_7]`.

4.2 The basic schema for two-headed rules

Our basic code generation schema generates:

1. One Prolog clause for each constraint symbol: this clause is executed when the constraint is posted to the constraint store; this clause makes the new constraint available as a partner and starts an active search for partners. Let `p/n` be a constraint symbol with partner occurrences `[part`$_1$`, ..., part`$_k$`]`. The following Prolog clause is generated for `p/n`:

```
p(X̄) :-
        gen_constr_id(Id),
        get_store(Store),
        Constr = constr_p(Id,Alive,X̄),
        partner(Store,Id,Alive,args(X̄)),
        post_events([part1, ... , partk],Store,Alive,Constr,args(X̄)).
```

2. One Action Rule (for the whole program) which sets up waiting partners:

```
partner(Store,Id,Alive,Args), var(Alive), {event(Store,Message),
                                                    ins(Alive)} =>
        Message = Occ - Constr,
        try_partner(Occ,Id,Alive,Args,Constr).
partner(_,_,_,_) => true.
```

3. One Prolog clause of the predicate `try_partner` for each occurrence: this clause represents a partner waiting to be visited by an active constraint. We deal with propagation rules only at first. Let the CHR rule be

$$\mathtt{p}(\bar{X}_p),\ \mathtt{q}(\bar{X}_q)\ \mathtt{==>}\ \mathtt{G}\ \mathtt{|}\ \mathtt{B}.$$

   Let the occurrence identifier of `p` in that rule be `p_k` and the rule identifier be `r_i`, then the `try_partner` clause for the occurrence of `p` in that rule is:

```
try_partner(p_k,Idp,Alivep,args(X̄p),constr_q(Idq,Aliveq,X̄q)) :-
        ( Idp \== Idq,
          var(Aliveq),
          not_in_history(Idp,Idq,r_i),
          G ->
              add_to_history(Idp,Idq,r_i),
              B
         ;
              true
         ).
```

   The order of these clauses is immaterial, as the predicate is always called with its first argument instantiated.

4. One Action Rule (plus some Prolog code) for initiating (and possibly retriggering) active search for partners:

```
post_events(Occurrences,Store,Alive,Constr,Args), var(Alive),
                                {generated, ins(Alive), ins(Args)} =>
      post_events(Occurrences,Store,Constr).
post_events(_,_,_,_,_) => true.

post_events([],_,_).
post_events([Occ|Occs],Store,Constr) :-
      post_event(Store,Occ-Constr),
      post_events(Occs,Store,Constr).
```

   When the constraint is created (as indicated by the event pattern `generated`) or when any variable in it is instantiated (as indicated by the event pattern `ins(arg(Args))`, the event representing the constraint `Constr` is posted to initiate search for partners. The refined semantics is preserved by posting the constraint to the channels of the occurrences in the specified order.

4.3 Dealing with Simplification and Simpation Rules

There is only one small change to the above schema for simplification and simpagation rules: when an occurrence is used in a `try_partner` clause, and the occurrence was in a position for removing the constraint (both in simplification, to the right of \ in simpagation), the corresponding `Live` variable is instantiated. For instance for the CHR rule

$$\texttt{p}(\bar{\texttt{X}}_p)\ \texttt{\textbackslash}\ \texttt{q}(\bar{\texttt{X}}_q)\ \texttt{==>}\ \texttt{G}\ \texttt{|}\ \texttt{B}.$$

and with all other assumptions as above, we get:

```
try_partner(p_k,Idp,Livep,args(X̄p),constr_q(Idq,Liveq,X̄q)) :-
       ( Idp \== Idq,
         var(Liveq),
         not_in_history(Idp,Idq,r_i),
         G ->
             Liveq = 1,
             add_to_history(Idp,Idq,r_i),
             B
        ;
             true
        ).
```

4.4 An Example: `primes.chr`

The CHR code for the prime benchmark is

```
prime(Y) \ prime(X) <=> 0 is X mod Y | true.
```

The occurrences of `prime` are numbered 1 (`prime(X)`) and 2 (`prime(Y)`). The specific code for this program is:

```
prime(X) :-
        gen_constr_id(Id),
        get_store(Store),
        Constr = constr_prime(Id,Alive,X),
        partner(Store,Id,Alive,args(X)),
        post_events([1,2],Store,Alive,Constr,args(X)).

try_partner(1,IdY,AliveY,args(Y),constr_prime(IdX,AliveX,X)) :-
        ( IdX \== IdY,
            var(AliveX),
            not_in_history(IdX,IdY,rule1),
            0 is Y mod X ->
            AliveY = 1,
            add_to_history(IdX,IdY,rule1)
        ;
            true
        ).
try_partner(2,IdX,AliveX,args(X),constr_prime(IdY,AliveY,args(Y))) :-
        ( IdX \== IdY,
            var(AliveY),
            not_in_history(IdX,IdY,rule1),
            0 is Y mod X ->
            AliveY = 1,
            add_to_history(IdX,IdY,rule1)
        ;
            true
        ).
```

4.5 Simple Specializations and Improvements of the Schema

In this section we add a number of simple, but well-known CHR optimizations to the schema and specialize the generic code for the individual constraint symbols.

Distributed Propagation History Rather than maintaining one central propagation history, the history can be distributed among the different constraint instances. This technique is used in the CHR references implementation by Holzbaur, and implemented by all CHR systems based on it. There are two advantages of the distributed representation: Firstly, the distributed parts are smaller and hence take less time to look for a particular tuple. Secondly, when a constraint is removed, its part of the history is removed as well.

For this the constriant event is extended with a new field *History*:

$$\texttt{constr}(\mathit{Cno}, \mathit{Alive}, \mathit{History}, \overline{X})$$

The data structure for this history is immaterial; it may well be an open-ended list.

The meaning of the two predicates for dealing with the history is also slightly altered:

- `not_in_history`(*History*, *PartnerNo*, *RuleNo*):
- `add_to_history`(*History*, *PartnerNo*, *RuleNo*): Let *History* be the history associted with a constraint. The primitive `not_in_history` succeeds if the tuple (*PartnerNo*, *RuleNo*) is not included in *History*, and the primitive `add_to_history` adds the tuple (*PartnerNo*, *RuleNo*) into *History*.

The constraint identifier of the active constraint is no longer explicitly store in the history, because every *History* implicitly corresponds to one particular constraint identifier.

Propagation History for Propagation Rules Actually, for simplification and propagation rules checking and updating the propagation history is not necessary. Because one or more of the involved constraints is removed in such a rule, it can never be executed more than once. Hence, these operations should only be done for propagation rules.

Constraint Store Indexing A similar idea of distribution applies to the constraint store. We partition it such that we do not have to look through the whole constraint store for potential partner constraints. This technique is called constraint store *indexing*.

In the context of action rules our constraint store is represented by the single `Store` channel. We partition it into multiple channel. For each occurrence of a constraint symbol, a channel is used for posting and receiving events. For each instance in the store of the corresponding constraint symbol, there is an agent attached to the channel. We need the following primitive:

- `get_channel`(*Id*, *Channel*): retrieves the channel for the occurrence with the identifier `Id`.[5]

A channel is assigned to an identifier when `get_channel` is called on it for the first time.

Note that we effectively replace in this way the manifest occurrence identifiers in the program with dedicated occurrence channels. This is more efficient than the general channel with the occurrence identifiers, because any event is broadcast to all agents on the channel. Hence, more selective channels are better.

[5] The channel of an occurrence is an attributed variable stored as a global heap variable. In B-Prolog, the built-in `global_heap_get(Name,Value)`, which is the same as `b_getval(Name,Value)` in h-Prolog and SWI-Prolog, is used to access global heap variables.

Specialization of Generic Code Rather than using the generic predicates and action rules, we now generated dedicated ones. This allows us to flatten some data structures. Firstly, we replace the generic `partner` agent and `try_partner` predicate with a single dedicated agent `agent_`P_i for every occurrence. Secondly we specialize the generic `post_events` agent into a dedicated `post_events_`p agent for every constraint symbol.

Our basic schema now generates the following Prolog clause for the symbol `p/n`:

$$
\begin{array}{l}
p(\overline{X}) :- \\
\quad \texttt{gen_constr_id}(Cno), \\
\quad Constr = constr(Cno, Alive, History, \overline{X}), \\
\quad \texttt{get_channel}(P_1, ChP_1), \ldots, \texttt{get_channel}(P_k, ChP_k), \\
\quad \texttt{get_channel}(Q_1, ChQ_1), \ldots, \texttt{get_channel}(Q_m, ChQ_m), \\
\quad \texttt{agent_}P_1(ChP_1, Cno, Alive, History, \overline{X}), \\
\quad \ldots \\
\quad \texttt{agent_}P_k(ChP_k, Cno, Alive, History, \overline{X}), \\
\quad \texttt{post_}p\texttt{_}n(ChQ_1, \ldots, ChQ_m, Alive, Constr, \overline{X}).
\end{array}
$$

For each occurrence identifier in the primal and partner occurrence lists, there is a call of `get_channel`, which gets the channel for the occurrence. For each primal occurrence P_i ($i = 1, \ldots, k$), there is a call named `agent_`P_i, which creates an agent for waiting for future partner constraints. The last call in the clause `post_`p`_`n posts the constraint $p(\overline{X})$ to the channels of the partner occurrences to initiate search for partners of $p(\overline{X})$.

Let the identifier of the rule in which P_i occurs be $RuleId$, the guard and body of the rule be G and B, respectively. If the rule is a propagation rule, then the agent for the occurrence P_i is defined as follows:

$$
\begin{array}{l}
\texttt{agent_}P_i(Ch, Cno, Alive, History, \overline{X}), \\
\quad \texttt{var}(Alive), \\
\quad \{\texttt{event}(Ch, Q), \texttt{ins}(Alive)\} \\
=> \\
\quad Q = constr(CnoQ, AliveQ, HistoryQ, \overline{Y}), \\
\quad (Cno \backslash == CnoQ, \\
\quad \texttt{var}(AliveQ) \\
\quad \texttt{not_in_history}(History, CnoQ, RuleId), \\
\quad \texttt{not_in_history}(HistoryQ, Cno, RuleId), \\
\quad G \; -> \\
\qquad \texttt{add_to_history}(History, CnoQ, RuleId), \\
\qquad \texttt{add_to_history}(HistoryQ, Cno, RuleId), \\
\qquad B \\
\quad ; \\
\qquad \texttt{true}). \\
\texttt{agent_}P_i(_, _, _, _, \overline{X}) => \texttt{true}.
\end{array}
$$

When the agent receives an event $constr(CnoQ, AliveQ, HistoryQ, \overline{Y})$, it checks the following: (1) the constraint represented by the event is different from the

constraint represented by this agent ($Cno\backslash==CnoQ$); (2) the constraint represented by the event is alive ($\texttt{var}(AliveQ)$); (3) the rule $RuleId$ has never been applied on $(Cno, CnoQ)$ or $(CnoQ, Cno)$; and (4) the guard G is satisfied. The body is executed when all these four conditions are satisfied. The histories associated with these two constraints are updated to record this application before the body is executed. Notice that the agent also watches the $\texttt{ins}(Alive)$ event. When the constrain represented by the agent is removed from the store ($Alive$ is bound to 0), the agent will be killed.

If the rule in which P_i occurs is a simplification rule, then the following action rule is generated for P_i:

$$
\begin{array}{l}
\texttt{agent_}P_i(Ch, Cno, Alive, History, \overline{X}), \\
\quad \texttt{var}(Alive), \\
\quad \{\texttt{event}(Ch, Q), \texttt{ins}(Alive)\} \\
=> \\
\quad Q = constr(CnoQ, AliveQ, HistoryQ, \overline{Y}), \\
\quad (Cno\backslash==CnoQ, \\
\quad \texttt{var}(AliveQ) \\
\quad G \; -> \\
\qquad Alive = 0, AliveQ = 0, \\
\qquad B \\
\quad ; \\
\qquad \texttt{true}).
\end{array}
$$

The status variables $Alive$ and $AliveQ$ are set to 0 to indicate that the constraints represented by the agent and the event have been removed from the store. If the rule is a simpagation rule, then only $Alive$ or $AliveQ$ is set to 0 depending on whether P_i occurs to the left or right of $\backslash$.

After all the agents $\texttt{agent_}P_1$, ..., $\texttt{agent_}P_k$ are created, an agent named $\texttt{post_}p_n$ is created to initiate search for partners.

$$
\begin{array}{l}
\texttt{post_}p_n(ChQ_1, \ldots, ChQ_m, Alive, Constr, \overline{X}) \\
\quad \texttt{var}(Alive), \\
\quad \{\texttt{generated}, \texttt{ins}(Alive), \texttt{ins}(\overline{X})\} \\
=> \\
\quad \texttt{post_event}(ChQ_1, Constr), \\
\quad \ldots \\
\quad \texttt{post_event}(ChQ_m, Constr). \\
\texttt{post_}p_n(ChQ_1, \ldots, ChQ_m, _, _, \overline{X}) => \texttt{true}.
\end{array}
$$

4.6 The Example Revisited

The following shows the new generated code for the `prime/1` program.

```
prime(X) :-
        gen_constr_id(Id),
        Constr = constr(Id,Alive,History,X),
```

```
        get_channel(prime_1_1,Ch1),
        get_channel(prime_1_2,Ch2),
        agent_prime_1_1(Ch1,Id,Alive,History,X),
        agent_prime_1_2(Ch2,Id,Alive,History,X),
        post_prime_1(Ch1,Ch2,Alive,Constr,X).

agent_prime_1_1(Ch,Id,Alive,History,X), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,Y),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            Alive = 0
        ;
            true
        ).
agent_prime_1_1(_,_,_,_,_) => true.

agent_prime_1_2(Ch,Id,Alive,History,Y), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,X),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            AliveQ = 0
        ;
            true
        ).
agent_prime_1_2(_,_,_,_,_) => true.

post_prime_1(Ch1,Ch2,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(Ch1,Constr),
        post_event(Ch2,Constr).
post_prime_1(_,_,_,_,_) => true.
```

5 Single-Headed Rules

Before we prevoide an efficient compilation schema for single-headed rules, we briefly consider a naive approach that reduces single-headed rules to two-headed ones.

5.1 Naive Approach

A simple solution to deal with single-headed rules, is to transform them into two-headed rules:

```
p <=> G | B.       to   dummy_partner \ p <=> G | B.
and
p ==> G | B.       to   dummy_partner, p ==> G | B.
```

and we assume a suitable initial addition of `dummy_partner` to the constraint store.

5.2 Improved Approach

The above naive solution to single-headed rules has the inconvenient requirement that `dummy_partner` has to be added initially. Moreover, it causes some undue overhead. Hence, we propose an alternative, no longer expressible at the level of CHR rules, that does not require such initialization.

For a single-headed CHR rule with the constraint symbol p/n in the head, a constraint of p/n does not need to wait for a partner constraint to trigger the rule. Let P_i be the occurrence identifier of the head. Each time a constraint of p/n is added into the store, an agent defined below is created:

$$\begin{array}{l}\texttt{agent_}P_i(PrivateCh, Cno, Alive, History, \overline{X}),\\ \quad \texttt{var}(Alive),\\ \quad \{\texttt{event}(PrivateCh, _),\texttt{ins}(Alive)\}\\ => \ldots\\ \texttt{agent_}P_i(_, _, _, _, \overline{X}) =>\texttt{true}.\end{array}$$

where the body of the action rule encodes the guard and body of the CHR rule. The agent is not activated by the posting of a partner constraint, but by the posting of the constraint itself. The event pattern $\texttt{event}(PrivateCh, _)$ means that no information from any event is used. By using a separate channel for each occurrence of p/n, we enforce as before the ordering of the rules in the refined semantics. However, now we have to use a private channel (one for each constraint instance rather than a global one) to enforce the depth-first execution order of the refined semantics. Otherwise we would break our assumption that a constraint only features as its own partner for a single-headed rule. The result would be a kind of breadth-first execution order.

Note that for single-headed rules the predicates that deal with the propagation history have one less argument. The partner constraint identifier is omitted.

5.3 An Example

Consider the following CHR program, (partially) implementing boolean `and/3` and `not/2` constraints:

```
and(X,X,Z) <=> Z = X.
not(X,Y) \ and(X,Y,Z) <=> Z = 0.
```

The following shows the generated code for `and/3`.

```
and(X,Y,Z) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,Y,Z),
        get_channel(and_3_2,PublicCh2),
        get_channel(not_2_1,ChNot),
        agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z),
        agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z),
        post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z).

agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PrivateCh1,_), ins(Alive)} =>
        ( X == Y ->
            Alive = 0,
            Z = X
        ;
            true
        ).
agent_and_3_1(_,_,_,_,_,_,_) => true.

agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PublicCh2,Constr), ins(Alive)} =>
        Constr=constr(Cno,AliveNot,HistoryNot,XNot,YNot),
        (Cno\==CnoNot,
         var(AliveNot),
         X == XNot,
         Y == YNot
          ->
            Alive = 0,
            Z = 0
        ;
            true
        ).
agent_and_3_2(_,_,_,_,_,_,_) => true.

post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z), var(Alive),
        {generated, ins(Alive), ins(X), ins(Y), ins(Z)} =>
        post_event(PrivateCh1,Constr),
        post_event(ChNot,Constr).
post_and_3(_,_,_,_,_,_,_) => true.
```

6 Multi-headed rules

For rules with more than two constraint patterns in the heads, we apply a similar binarization technique as in the famous RETE algorithm [12] for production systems. This technique makes the intermediate joins of partner constraints manifest in temporary constraints.

Conceptually for a rule with $n \geq 2$ constraint patterns:

$$p_1(\overline{X}_1), \ldots, p_n(\overline{X}_n) ==> G \mid B.$$

We compile it into $(n-1)$ two-headed rules with $(n-1)$ new constraint symbols $\mathtt{p}_{1\ldots j}$ where $2 \leq j \leq (n-1)$:

$$\begin{array}{l} p_1(\overline{X}_1), p_2(\overline{X}_2) ==> p_{1..2}(\overline{X}_1, \overline{X}_2). \\ p_{1..2}(\overline{X}_1, \overline{X}_2), p_3(\overline{X}_3) ==> p_{1..3}(\overline{X}_1, \overline{X}_2, \overline{X}_3). \\ \vdots \\ p_{1..(n-1)}(\overline{X}_1, \ldots, \overline{X}_{(n-1)}), p_n(\overline{X}_n) ==> p_{1..n}(\overline{X}_1, \ldots, \overline{X}_n). \end{array}$$

It has often been claimed informally that the correct binarization of CHR rules can be expressed trivially as the above source-to-source transformation is correct. In effect, early implementations of CHR did not support any n-headed rules where $n > 2$, partially for that reason.

In reality, the binarization is not as simple as that, and we believe that it cannot be expressed fully as a source-to-source transformation. A number of issues arise that have to be dealt with at a lower level.

Firstly, no head constraint can be used twice in the matching against the constraint patterns in the original CHR rule. To ensure this, we let each temporary constraint carry the identifiers of the precedence constraints from which it is obtained. Each time before we fire the rule $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j), p_{j+1}(\overline{X}_{j+1}) ==> \ldots$, we check if $p_{j+1}(\overline{X}_{j+1})$ occurs in the precedence constraints that led to the generation of $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j)$. If so, the rule cannot be fired.

Secondly, the following relationship among the status variables must be maintained: whenever a precedence constraint is removed from the store, the temporary constraints created from it must be removed as well. For this reason we let each temporary constraint carry in a similar fashion the status variables of the precedence constraints. On an `ins` event of any of these status variables the agents for the temporary constraints are removed.

Thirdly, for simplification and simpagation rules, it should be possible at the end of the join chain, before executing the body of the original rule, to remove the original precedence constraints. For this purpose, the status variables of the precedence constraints must be carried by the temporary constraints. Luckily, this is already the case for the previous issue.

Finally, the temporary constraints should be properly synchronized with their precedence constraints. If a variable appearing in a precedence constraint is instantiated, any temporary constraint that caries that variable should not fire its event, before the events for the previous occurrences of the precedence constraint do. Our solution is to not fire events for temporary constraints upon `ins` events of the temporary constraints' variables. Instead every precedence constraint has a private channel for each multi-headed occurrence. This channel is carried by the temporary constraints. Upon an `ins` event for the precedence constraint, first the usual events are posted for the preceding occurrences before an event is posted to this private channel. The temporary constraint react on this event by firing their usual events.

6.1 An Example

Consider the following articifial CHR program:

```
a(X) \ b(X), c(X) <=> X = 42.
```

The following shows the generated code for `a/1` and the temporary constraint `temp12/8`.

```
a(X) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,ChTemp),
        get_channel(a_1_1,Ch1),
        get_channel(b_1_1,ChB),
        agent_a_1_1(Ch1,Cno,Alive,History,X,ChTemp),
        post_a_1(ChB,ChTemp,Alive,Constr,X,Y,Z).

agent_a_1_1(Ch,Cno,Alive,History,X,ChTemp), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(CnoB,AliveB,HistoryB,XB,ChTempB),
        (Cno\==CnoB,
         var(AliveB),
         not_in_history(History,CnoB,r1),
         not_in_history(HistoryB,Cno,r1)
          ->
            add_to_history(History,CnoB,r1),
            add_to_history(HistoryB,Cno,r1),
            temp12(X,XB,Cno,CnoB,Alive,AliveB,ChTemp,ChTempB)
        ;
            true
        ).
agent_a_1_1(_,_,_,_,_,_) => true.

post_a_1(ChB,ChTemp,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(ChB,Constr),
        post_event(ChTemp,Constr).
post_a_1(_,_,_,_,_) => true.

temp12(XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTempB) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,XA,XB,...),
        get_channel(temp12_1,Ch1),
        get_channel(c_1_1,ChC),
        agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,...),
        post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB).

agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTempB),
```

```
        var(Alive),var(AliveA),var(AliveB),
        {ins(Alive),ins(AliveA),ins(AliveB),event(CH1,Constr)} =>
        Constr=constr(CnoC,AliveC,HistoryC,XC,ChTempC),
        (Cno\==CnoC, CnoA\==CnoC, CnoB\==CnoC,
         var(AliveB),
         XA==XB, XB==XC
         ->
            AliveB=0, AliveC=0,
            XA = 42
        ;
            true
        ).
agent_temp12_8_1(_,_,_,_,_,_,_,_,_,_,_,_,_) => true.

post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB),
        var(Alive),var(AliveA),var(AliveB),
        {generated,ins(Alive),ins(AliveA),ins(AliveB),
         event(ChTempA,ChTempB} =>
        post_event(ChC,Constr).
post_temp12_8(_,_,_,_,_,_,_) => true.
```

6.2 Simple Improvements

Channel Argument Reduction The number of arguments of the agents grows quite quickly for large heads. There is one `ChTemp` for every head constraint. It is possible to reduce this amount by *combining* several of these variables into one.

Firstly multiple `ChTemp` channels can be relayed through a new channel. For example, we have two channels `ChTemp1` and `ChTemp2`. We make a new channel `ChTempNew` and relay the events of the two channels through it with `relay_event(ChTemp1,ChTempNew, relay_event(ChTemp2,ChTempNew)` where `relay_event/2` is defined as:

```
relay_event(Ch1,Ch2), var(Ch1), {ins(Ch1), event(Ch2,Event)} =>
     post_event(Ch1,Event).
relay_event_event(_,_) => true.
```

`Alive` Argument Reduction A similar approach is possible to reduce the number of `Alive` variables. In this case, it is the instantiation that is to be relayed. However, `Alive` variables do not only signal instantation to the partial joins. Those of the removed heads are also instantiated when the rule applies. For this reason, we will join toghether the `Alive` variables of removed and kept heads separately and instantiation of the new `Alive` variable for the removed heads is relayed back to the individual alive variables.

For example, when the rule head is `a, b \ c, d` with respective variables `AliveA`, `AliveB`, `AliveC` and `AliveD`, then two new variables `AliveKept` and `AliveRemoved` are used that are connected with the other ones by:

```
        relay_inst(AliveA,AliveKept),
        relay_inst(AliveB,AliveKept),
        relay_inst(AliveC,AliveRemoved),
        relay_inst(AliveD,AliveRemoved),
        relay_inst(AliveRemoved,AliveC),
        relay_inst(AliveRemoved,AliveD)
```

The agent `relay_inst/2` is defined as:

```
relay_inst(Alive1,Alive2), var(Alive1), {ins(Alive1)} =>
        Alive2 = 1.
relay_inst(_,_) => true.
```

Early Guard Scheduling In many cases the guard, or part of it, can already be evaluated in a partial rather than a full join of all the heads. By scheduling it as soon as possible, failure of the guard can be detected much earlier and much unnecessary work is avoided.

The guard tests that can be scheduled earlier are those whose success and failure cannot change over time. This concerns in particular tests over ground terms. As the ground terms do not change over time, no test over them can change its truth value.

For example, assume the head `a(X), b(Y), c(Z)` with guard `X < Y` where we know that `X` and `Y` are ground. In this case we can schedule the test as soon as we have the partial join `a(X), b(Y)`.

7 Optimizations

The above compilation schema is quite basic in a number of ways. This section discusses a few optimizations that greatly improve the efficiency. We focus mainly on adaptations of existing optimizations [17, 20] for ARs.

7.1 Indexing

A large portion of execution time is spent looking for partners for posted constraints. The time needed to look for a partner constraint is determined by the number of agents attached to the channel for the partner occurrence. In the basic schema, that number is equal to the number of constraints of the partner symbol in the store. Indexing can reduce the number of constraints to be searched.

For a double-headed rule with two constraint patterns $p(X_1, \ldots, X_i, \ldots, X_n)$ and $q(Y_1, \ldots, Y_m)$, and an equality argument test $X_i = T$ in the guard,[6] the constraints of p/n can be indexed on X_i if T is a nonvariable term or an argument Y_j of the partner pattern. Hence, when a constraint of q/m is posted only those constraints of p/n whose ith argument is equal to T need to be searched. We call the argument X_i an *index argument*. Each index argument has a unique

[6] Recall all rules are in the head normal form.

identifier which is made up of the identifier of the constraint pattern in which it occurs and the number of the argument.

For each index argument, we use a special channel for it and use the following primitive to get the channel:

- `get_channel`$(CNo, X, Channel)$: retrieves the channel for the index argument with the identifier CNo and value X.

Because X may contain variables and thus may change during execution, it must be guaranteed that all arguments of constraints with the same identifier and the same value share the same channel.

In general, head constraints can be indexed on multiple arguments if there are multiple equality argument tests in the guard. Let r be a CHR rule with a constraint pattern $p(X_1, \ldots, X_n)$, a partner constraint pattern $q(Y_1, \ldots, Y_m)$, and the following equality argument tests in the guard:

$$X_{i_1} = T_1, \ldots, X_{i_l} = T_l \text{ where } 1 \leq i_1, \ldots, i_l \leq n$$

where each $T_k (1 \leq k \leq l)$ is a nonvariable term or an argument in the partner constraint pattern. If T_k is equal to some argument Y_{j_k} in the partner constraint, then X_{i_k} and Y_{j_k} are called *shared index arguments* in the constraint patterns. For each index argument $X_{i_k} (1 \leq k \leq l)$, we use a special channel for it, to which all the constraints of p/n whose i_kth argument is equal to T_k are attached.

Let P be the identifier of the occurrence $p(X_1, \ldots, X_n)$ in rule r, and let $ChP_{X_{i_1}}, \ldots, ChP_{X_{i_l}}$ be the special channels for the index arguments $X_{i_1}, \ldots, X_{i_l}$. For each constraint $p(\overline{X})$ of p/n, the agent created for the occurrence takes the list of these channels and watches partner constraints posted to any of these channels:

```
agent_P(Cno, ChP_{X_{i_1}}, ..., ChP_{X_{i_l}}, Alive, History, X̄),
    var(Alive),
    {event(ChP_{X_{i_1}}, Constr), ...,event(ChP_{X_{i_l}}, Constr),ins(Alive)}
=> ...
```

The constraints of the occurrence $p(X_1, \ldots, X_n)$ connected to one channel in the original basic translation schema is divided into several (possibly overlapping) groups each of which has a separate channel.

Suppose the arguments $Y_{j_1}, \ldots, Y_{j_k}$ in the partner pattern are shared index arguments (i.e., there are tests $X_{i_1} = Y_{j_1}, \ldots, X_{i_k} = Y_{j_k}$ in the guard). Let $QY_{j_1}, \ldots, QY_{j_k}$ be the identifiers, and $ChQ_{Y_{j_1}}, \ldots, ChQ_{Y_{j_l}}$ be the special channels for the index arguments. When a constraint $p(\overline{X})$ is posted, we only need to search the constraints of q/m that are connected to all the channels: $ChQ_{X_{i_1}}, \ldots, ChQ_{X_{i_k}}$. To achieve this, we change the definition of `post_p_n` to the following:

```
post_p_n(ChQ_{Y_{j_1}}, ..., ChQ_{Y_{j_k}}, Alive, Constr, X̄)
    var(Alive),
    {generated,ins(Alive),ins(X̄)}
=>
```

..., % for partner occurrences before $q(Y_1, \ldots, Y_m)$
`get_channel`$(QY_{j_1}, X_{i_1}, ChQ_{Y_{j_1}})$,
...,
`get_channel`$(QY_{j_k}, X_{i_k}, ChQ_{Y_{j_k}})$,
`post_event`$(ChQ_{Y_{j_1}} /\backslash \ldots /\backslash ChQ_{Y_{j_k}}, Constr)$,
.... % for partner occurrences after $q(Y_1, \ldots, Y_m)$
`post_`$p_n(_, _, \overline{X})$ =>`true`.

where the channels are combined using `post_event(Index1/\.../\Indexn,Event)`, the new feature that only posts the event to agents that are attached to *all* of the channels `Index1,...,Indexn`. The channels are refreshed before posting the event since some of the index arguments may have changed since the last posting.

Consider, for example, the following CHR program:

```
p(X,Y),q(Y,X)<=>true.
```

The identifiers for `X` and `Y` in `p(X,Y)` are `p_2_1_1` and `p_2_1_2`, respectively, and the identifiers for `Y` and `X` in `q(Y,X)` are `q_2_1_1` and `q_2_1_2`, respectively. The following shows the compiled code for `p/2`:

```
p(X,Y) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,Y),
        get_channel(p_2_1_1,X,ChP1),
        get_channel(p_2_1_2,Y,ChP2),
        get_channel(q_2_1_1,Y,ChQ1),
        get_channel(q_2_1_2,X,ChQ2),
        agent_p_2_1(Cno,ChP1,ChP2,Alive,History,X,Y),
        post_p_2(ChQ1,ChQ2,Alive,Constr,X,Y).

agent_p_2_1(Cno,ChP1,ChP2,Alive,History,X,Y),var(Alive),
        {event(ChP1,Constr), event(ChP2,Constr), ins(Alive)}
=>
        Constr=constr(CnoQ,AliveQ,HistoryQ,Y1,X1),
        (Cno\==CnoQ,
         var(AliveQ),
         X==X1,Y==Y1 ->
            Alive = 1, AliveQ = 1
        ;
            true
        ).
agent_p_2_1(_,_,_,_,_,_,_,_,_) => true.

post_p_2(ChQ1,ChQ2,Alive,Constr,X,Y),var(Alive),
        {generated, ins(Alive), ins(X), ins(Y)}
=>
        get_channel(q_2_1_1,Y,ChQ1),
        get_channel(q_2_1_2,X,ChQ2),
```

```
        post_event(ChQ1/\ChQ2,Constr).
post_p_2(_,_,_,_,_,_) => true.
```

7.2 Combining channels and agents.

For a rule with two constraint patterns of the same symbol, if the ordering of the two patterns is not important, we use only one channel for them and create only one agent for each constraint of the symbol. For example,

```
        leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

As the ordering of the two constraint patterns is not important, we create an agent of the following when a constraint of `leq/2` is added into the store:

```
agent_leq_2_12(Cno,Ch,Alive,History,X,Y),var(Alive),
        {event(Ch,Constr),ins(Alive)}
=>
        Constr=constr(CnoQ,AliveQ,HistoryQ,U,V),
        (Cno\==CnoQ,
         var(AliveQ),
         not_in_history(History,CnoQ,r1),
         not_in_history(HistoryQ,Cno,r1)
         (Y==U->
            add_to_history(History,CnoQ,r1),
            add_to_history(HistoryQ,Cno,r1),
            leq(X,V)
         ;V==X->
            add_to_history(History,CnoQ,r1),
            add_to_history(HistoryQ,Cno,r1),
            leq(U,Y)
         ;true).
agent_leq_2_12(_,_,_,_,_,_) => true.
```

7.3 Single-headed rules.

We can merge the agents of multiple consecutive single-headed rules and their corresponding private channels into a single agent and a single private channel. The bodies of the rules are merged appropriately in a single body. Subsequent bodies of a propagation rule are put in conjunction (i.e. any continuation), and of simplification rules are put in the else-branch (i.e. fail continuation) of that body.

Consider a consecutive chunck of single-headed rules:

$p(\overline{X}) => G_i|B_i.$
$\ldots$
$p(\overline{X}) => G_j|B_j.$

Instead of using one private channel for each constraint for each rule, we use one defined as follows for each constraint for the entire chunk:

$\texttt{agent_}p\texttt{_}n\texttt{_}ij(Cno, PrivateCh, Alive, History, \overline{X})$,
    $\texttt{var}(Alive)$,
    $\{\texttt{event}(PrivateCh, _),\texttt{ins}(Alive)\}$
$=>\ldots$
    $\ldots$, % other condition
    $(G_i -> B'_i; \ldots; G_j -> B'_j; true)$.
$\texttt{agent_}p\texttt{_}n\texttt{_}ij(_, _, _, _, \overline{X}) =>\texttt{true}$.

where each B'_k encodes the body B_k and the calls for instantiating the status variable (if it is a simplification) and updating the history (if it propagation rule).

For a constraint symbol that is defined by only single-headed rules, no private channel is needed at all to control the ordering.

7.4 Never-triggering constraints.

A constraint symbol whose instances do not *trigger*, e.g. because they are fully instantiated when called, does not require its agents to watch `ins/1` events of the arguments.

8 Evaluation

We have written an automatic translator from CHR to Action Rules, following the above basic compilation schema. This compiler is evaluated on a representative set of benchmarks from [19]. In addition, to get a better feeling for the attainable efficiency, we have also hand-generated code for the smallest benchmarks. The compiler, benchmarks and hand-optimized code are all available from the webpage `http://www.probp.com/chr/`.

We have used B-Prolog 6.9-b5. As a reference for comparison, we take the K.U.Leuven CHR system [21] in SWI-Prolog 5.6.11 and a port of it to B-Prolog. K.U.Leuven CHR is currently the most optimized CHR-system for Prolog. Tabel 1 lists the timings in milliseconds, obtained on a Pentium 4 1.8 GHz.

In order to interpret the figures in Tabel 1 correctly, one must take into account that B-Prolog is on average about 5 to 10 times faster than SWI-Prolog. For the case of attributed variables, it appears that the factor is more in the order of 2 to 3.

Unsurprisingly, the basic schema's performance is not so good. However, the hand-optimized code shows that drastic improvements are possible. In fact, such improvements are capable of considerably outperforming the K.U.Leuven CHR system in its current form. This suggests a worthwhile set of optimizations that can be generalized to other compilation schema, such as that of the K.U.Leuven CHR system.

We cannot reliably experiment with multi-headed rules that involve **Reactivate** transitions yet. The Action Rules semantics currently allows for a delay

Benchmarks	B-Prolog			SWI-Prolog
	Basic Compiler	Hand-Optimized	K.U.Leuven CHR	K.U.Leuven CHR
`fib (22)`	3,342	159	1,228	3,270
`fulladder (6000)`	995	-	180	350
`leq (50)`	73,218	178	1,073	3,160
`leq2 (50)`	75,253	183	1,738	4,060
`primes (2500)`	1,661	913	1,846	7,270
`wfs (1000)`	11,520	-	1,985	5,380
`zebra (10)`	3,318	630	2,390	7,220

Table 1. Benchmark Timings

between an instantiation and its corresponding `ins/1` event, where other code can be run in between that is out of order with respect to CHR's refined semantics. We will also address this issue in a future version.

9 Conclusion

In this paper we have presented a novel schema for compiling CHR to Action Rules. On the one hand, the AR schema has some elegance over the attributed variables schema in the expression as it allows some loops to be expressed implicitly, just like CHR. On the other hand, we have experienced some difficulty with strictly following the refined operational semantics of CHR; it is not such a good match for the operational semantics of AR.

Our initial experimental results show that the basic performance is rather bad, but hand-optimizations give very encouraging results and suggest worthwhile optimizations for all CHR systems. Another important result of our schema is that it shows how to properly binarize multi-headed rules, and that this is non-obvious. This shows the advantages of the attributed variables schema over the RETE-based approach. In [7] it was already shown that RETE's time and space behavior is worse than that of LEAPS [6], by which the attributed variables implementation of CHR was inspired.

A current limitation of the Action Rules schema is that it does not fully support an extension of CHR, called CHR$^\vee$ [3], which allows disjunctions in the bodies of rules. An Action Rule behaves as if the body is contained inside the ISO-Prolog `once/1` built-in: after executing the body, any remaining choice points created during the execution are pruned.

In future work, we will extend and automate the proposed optimizations. In particular, we will focus on Action Rules specific optimizations. A hybrid compilation schema seems another worthwhile topic for further research: selecting the most efficient parts of the attributed variables and Action Rules schemas, possibly based on properties of the program being compiled. This could also allow us to lift the current restriction of the Action Rule schema and fully support CHR$^\vee$. Finally, we will investigate were the strict adherence to the refined operational semantics can be lifted without harm.

References

1. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001.
2. Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.
3. Slim Abdennadher and Heribert Schütz. $\text{CHR}^{\vee}$: a flexible query language. In *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.
4. Marco Alberti, Federico Chesani, Alessio Guerri, Marco Gavanelli, Evelina Lamma, Paola Mello, Michela Milano, and Paolo Torroni. Expressing interaction in combinatorial auction through social integrity constraints. In Armin Wolf, Thom Frühwirth, and Marc Meister, editors, *W(C)LP'05: Proceedings of 19th Workshop on (Constraint) Logic Programming*, pages 53–64, Ulm, Germany, 2005.
5. Marco Alberti, Anna Ciampolini, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Logic Based Semantics for an Agent Communication Language. In *FAMAS 2003: In Proceedings of the International Workshop on Formal Approaches to Multi-Agent Systems*, pages 21–36, Warsaw, Poland, 2003.
6. Don Batory. The leaps algorithm. Technical report, Austin, TX, USA, 1994.
7. David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 287–296. Morgan Kaufmann, 1991.
8. Henning Christiansen. CHR Grammars. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):467–501, 2005.
9. Emmanuel Coquery. TCLP: A type checker for CLP($\mathcal{X}$). In Fred Mesnard and Alexander Serebrenik, editors, *Proceedings of the 13th Workshop on Logic Programming Environments*, pages 17–30. Katholieke Universiteit Leuven, 2003.
10. Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *CP'99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 174–188, Alexandria, Virginia, USA, 1999. Springer Verlag.
11. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, September 2004. Springer Verlag.
12. Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
13. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
14. Thom Frühwirth and Pascal Brisset. Optimal Placement of Base Stations in Wireless Indoor Telecommunication. In Michael J. Maher and Jean-Francois Puget, editors, *CP'98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 476–480, Pisa, Italy, October 1998. Springer Verlag.
15. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.

16. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
17. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):503–531, 2005.
18. IC-Parc. ECL^{i}PS^e. http://www.icparc.ic.ac.uk/eclipse/.
19. Tom Schrijvers. A Collection of Assorted CHR Benchmarks, 2005. http://www.cs.kuleuven.be/~toms/Research/CHR/.
20. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2005. xxiv+210 pages.
21. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004.
22. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.
23. Michael Thielscher. FLUX: A Logic Programming Method for Reasoning Agents. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):533–565, 2005.
24. Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *CP'01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 2239, page 256. Springer Verlag, January 2001.
25. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *To appear in Theory and Practice of Logic Programming (TPLP)*, 2006.