

Aggregates for CHR through Program Transformation

Peter Van Weert*, Jon Sneyers**, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. We propose an extension of Constraint Handling Rules (CHR) with aggregates such as `sum`, `count`, `findall`, and `min`. This new feature significantly improves the conciseness and expressiveness of the language. In this paper, we describe an implementation based on source-to-source transformations to CHR (extended with some low-level compiler directives). We allow user-defined aggregates and nested aggregate expressions over arbitrary guarded conjunctions of constraints. Both an on-demand and an incremental aggregate computation strategy are supported.

1 Introduction

Constraint Handling Rules (CHR) [1, 4] is a powerful, elegant committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the implementation of constraint solvers, CHR has matured towards a general purpose language, used in a wide range of application domains, including natural language processing, multi-agent systems, and type system design.

In [8, 9] we proposed an extension of CHR with *aggregates*. This declarative language feature allows the aggregation of information from an unbounded number of constraints to be captured concisely in a single expression in the head of a CHR rule. Example aggregates are `sum`, `count`, `findall`, and `min`. Without language support for aggregates, these common programming idioms require cumbersome, low-level auxiliary constructs, cross-cutting the entire program. Case studies show aggregates reduce program size by up to 50%. The resulting programs are also significantly more understandable, maintainable, and robust.

This paper presents how existing CHR systems can be extended with a general, extensible aggregate framework using source-to-source transformations to lower-level CHR. Only a small number of easily implemented low-level compiler directives have to be added to the CHR system itself. The transformation takes care of introducing auxiliary and cross-cutting code, not unlike an aspect weaver in Aspect-Oriented Programming [5].

The source-to-source transformation schemes presented in this paper support user-defined, application-tailored aggregates, nested aggregate expressions, and

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

** Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

efficient aggregate computation using either on-demand or incremental aggregate computation. The design of these non-trivial transformation schemes is discussed in detail, the different issues identified and addressed one by one.

Overview. Section 2 briefly recalls the syntax and operational semantics of CHR. More information can be found in [3, 4]. Section 3 motivates and introduces the extension of CHR with aggregates. Next, two different source-to-source schemes are presented in Section 4. The implementation approach is evaluated in Section 5. Finally, Section 6 provides conclusions and directions of future work.

2 Preliminaries: Constraint Handling Rules

2.1 Syntax of CHR

A constraint $c(x_1, \dots, x_n)$ is an atom of predicate c/n , with all x_i values of a host language data type. Two types of constraints exist: *built-in constraints*, solved by an underlying solver, and *CHR constraints*, solved by the CHR program.

A CHR program consists of a sequence of CHR rules of the form:

$$\textit{name} @ H_k \setminus H_r \iff G \mid B$$

The *name* is optional and unique; rules without a name get one implicitly. The *head* consists of two conjunctions of CHR constraints, H_k and H_r . Their conjuncts are called *occurrences* (*kept* and *removed occurrences* resp.). The *guard* G is a conjunction of built-in constraints. If “ $G \mid$ ” is omitted, it is considered to be “*true* \mid ”. The *body* B is a conjunction of CHR and built-in constraints.

There are three types of rules. If H_k is empty, the rule is a *simplification* rule. If H_r is empty, the rule is a *propagation* rule and the symbol “ \implies ” is used instead of “ \iff ”. If both parts are non-empty, the rule is a *simpagation* rule. At least one of H_r and H_k must be non-empty.

Logically, a simplification rule corresponds to an equivalence: $G \rightarrow (H_r \leftrightarrow B)$, while a propagation rule corresponds to an implication: $G \rightarrow (H_k \rightarrow B)$.

2.2 Operational Semantics of CHR

Informally, the operational semantics of a CHR rule is as follows: if for each occurrence in the head a matching constraint is found in the *constraint store*, and the guard is satisfied, then the rule *fires*: the constraints that matched the removed occurrences (H_r) are deleted from the store and the body is executed.

Formally, the execution of a CHR program follows the *theoretical* or *high-level* operational semantics, denoted as ω_t . For brevity, we do not present the formal transition rules of ω_t here; we refer to [3, 4] instead. A version of ω_t extended with aggregates is presented in Section 3.3.

The *theoretical* operational semantics is highly nondeterministic. Only programs that do not depend on the order of rule application have guaranteed

behavior under ω_t . Such programs are called *confluent* (cf. [4]). However, writing confluent programs is often overly difficult. Many programs are non-confluent under ω_t as CHR programmers exploit the execution strategy implemented by most CHR systems to obtain the desired behavior. The *refined operational semantics*, denoted with ω_r , instantiates ω_t to capture the behavior of most current systems. A complete exposition, including a formal description, is found in [3].

A central concept in the refined semantics is the *active constraint*. Each time a constraint becomes active, all CHR rules are tried in a *top-down textual order*, until all applicable rules that match the active constraint have been executed, or the active constraint is removed. If a rule fires, the constraints in its body are processed one at a time, in a *left-to-right textual order*. If a CHR constraint is processed, it is added to the constraint store and immediately becomes the new active constraint. Processing a built-in constraint entails solving it, and reactivating all CHR constraints whose arguments are affected, one at a time. The order in which CHR constraints are reactivated is undetermined. The activation and reactivation of a CHR constraint is treated as a *procedure call*: only when its execution is finished, the execution returns to the previous active constraint.

3 Extending CHR with Aggregates

As CHR is already Turing complete [7], aggregates do not add to the computational power of CHR. Section 3.1 shows they are nevertheless invaluable when it comes to expressiveness, maintainability and conciseness. The extension of CHR with aggregates is introduced in Section 3.2, and given a formal operational semantics in Section 3.3. A more thorough introduction to the proposed extension, more examples and case studies can be found in [8, 9].

3.1 Motivation and Running Example

As the head of each CHR rule only considers a fixed number of constraints, any form of aggregation over unbounded parts of the constraint store necessarily requires explicit encoding, using auxiliary constraints and rules. The following example clearly shows the inadequacy of such ad hoc approaches. It is also used as a running example throughout the paper.

Example 1. Suppose the constraints `account(AccountId,ClientId,Balance)` and `client(ClientId)` constitute a simplified representation of the accounts and clients of a bank. At some point, the bank decides to add the business rule:

“A client whose accumulated sum of account balances is \$25,000 or more is a *platinum client*”

As a client can have any number of accounts, this seemingly simple rule cannot be expressed straightforwardly in CHR. CHR practitioners therefore commonly introduce a constraint such as `accumulated_balance/2`. This allows the logic of platinum clients to be captured concisely in a single rule as follows:

```
client(C), accumulated_balance(C,Sum) ==> Sum ≥ 25000 | platinum(C).
```

This approach, however, also necessitates the explicit maintenance of the accumulated balance. This inherently cross-cutting concern requires invasive modifications to all rules that alter the balance of an account. The bank e.g. has to add at least the following underlined code:

```
deposit(A,X), account(A,C,B), accumulated_balance(C,Acc) <=>
    account(A,C,B+X), accumulated_balance(C,Acc+X).
...
withdraw(A,X), account(A,C,B), accumulated_balance(C,Acc) <=>
    B > X, account(A,C,B-X), accumulated_balance(C,Acc-X).
```

Many variations to the above maintenance scheme can be concocted, but they all require similar modifications scattered throughout the entire program. Similar auxiliary code has to be written for *every* aggregate; a very tedious and repetitive task. Clearly, this approach displays poor compliance with common software quality criteria: it is highly error-prone, and it impairs the readability and maintainability of the program, as the logic of many rules becomes tangled with obfuscating auxiliary code. In other words, many practical advantages of declarative programming – understandability, maintainability, robustness, and shortened development time – are severely handicapped.

3.2 An Extensible Framework for Aggregates in CHR

This section introduces an extension of CHR with aggregates, designed to overcome the expressivity problems outlined in the previous section. It allows rule heads to contain *aggregates*. These expressions accumulate information over possibly unbounded parts of the constraint store. Aggregates can be written in both the kept and the removed part of the head; there is no semantical difference.

This section provides a short summary on the proposed, extensible aggregate framework. More information can be found in [8, 9].

Predefined aggregates. Our framework provides a wide range of predefined aggregates, including all aggregates commonly found in related paradigms such as database query languages [10] (i.e. `min`, `max`, `sum`, `count` and `avg`) and production rule systems (i.e. `not`, `exists` and `forall`). A complete list of predefined aggregates, together with a number of example uses, can be found in [8, 9].

Example 2. Using the `sum` aggregate (in italics), the platinum client business rule of Example 1 is again declaratively expressed in a single rule:

```
client(C), sum(B,account(_,C,B),Sum) ==> Sum ≥ 25000 | platinum(C).
```

However, no further changes to the program are required, as the aggregate’s semantics already guarantees the correct behavior implicitly: it accumulates the sum of the balances `B` of all matching `account/3` constraints, and ensures that the rule fires as soon as this sum, `Sum`, reaches 25,000.

Contrasting the above example with the approach outlined in Example 1 in the previous section clearly shows that aggregates render CHR programs more declarative, readable and maintainable. Relieved from the cumbersome and repetitive task of implementing aggregates, the programmer can focus exclusively on the application domain. So, productivity is improved considerably as well.

User-defined aggregates. Often information has to be aggregated in application-specific ways. Therefore, we designed a general high-level mechanism that enables CHR end-users to create *user-defined aggregates*:

```
aggregate(Start, Inc, Dec, Final, Template, Goal, Result)
```

The `aggregate/7` construct is expressive enough to effectively specify any aggregate. In fact, all predefined aggregates are also implemented by it.

Example 3. The predefined `sum(T,G,R)` aggregate for instance is specified as `aggregate(=(0),plus,minus,=,T,G,R)`, where `'=(0)'` indicates unification with zero, and `plus/3` and `minus/3` are two straightforward Prolog predicates computing the sum, respectively the difference of the first two arguments.

The first four arguments of `aggregate/7` specify the host language procedures or CHR constraints that determine how the aggregate is computed. First, an intermediate working value is *initialized* using `Start`. Then, for each matching found for `Goal`, a corresponding instance of `Template` is passed to `Inc` to *increment* the current working value. After all increments required are made, the working value is *finalized* using `Final`, to obtain the aggregate's result `Result`. The function of `Dec` is explained in Section 4.3.

These seven arguments thus completely determine an aggregate's semantics, as also reflected in the formal operational semantics presented in the next section.

Complex aggregate goals. Example 2 showed an aggregate over a simple `Goal`, i.e., consisting of a single CHR constraint. The *aggregate goal* `Goal` however can be an arbitrary conjunction of CHR constraints and guards: for example `count((platinum(C),account(_,C,_)), N)` counts the number of accounts owned by platinum clients. We further allow *nested aggregates*, that is, aggregate expressions inside the goal of another aggregate: for instance, `max(S, (client(C), sum(B,account(_,C,B),S)), M)` returns the largest total balance `M` of any individual client.

3.3 Formal Operational Semantics

We extend the theoretical operational semantics ω_t [3, 4] to deal with the general `aggregate/7` expressions introduced in the previous section (recall that this also covers all predefined aggregates). The extended semantics is denoted ω_a . We extend ω_t because of brevity, and because it allows more implementation freedom then extending a more deterministic instance such as e.g. ω_r (cf. Section 2.2).

The ω_a semantics is formulated as a state transition system. Transition rules define the relation between an execution state and its subsequent execution state.

Definition 1 (Identified constraints). To differentiate amongst otherwise identical copies of constraints, CHR constraints are assigned unique identifiers. An identified CHR constraint with constraint identifier i is denoted $c\#i$. We further introduce the functions $\text{chr}(c\#i) = c$ and $\text{id}(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner.

Definition 2 (Execution state). An execution state σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The goal \mathbb{G} is a multiset of constraints. The CHR constraint store \mathbb{S} is a set of identified CHR constraints (while \mathbb{S} is a set, $\text{chr}(\mathbb{S})$ is a multiset). The built-in constraint store \mathbb{B} is the conjunction of all built-in constraints passed to the underlying solver. The propagation history \mathbb{T} , necessary to prevent trivial non-termination, is a set of tuples, each recording the name of a rule and a sequence of identities of the CHR constraints that fired that rule. Finally, the counter $n \in \mathbb{N}$ represents the next unique constraint identifier.

The semantics of the built-in constraints is determined by a constraint theory $\mathcal{D}_{\mathcal{B}}$. Let $\text{vars}(A)$ be the variables occurring freely in A , then $\exists_A F$ denotes $\exists x_1, \dots, \exists x_n F$, with $\{x_1, \dots, x_n\} = \text{vars}(F) \setminus \text{vars}(A)$.

Because aggregates can be nested, we use two mutually recursive definitions:

Definition 3 (Matching substitutions). Let $\text{matchings}(A \wedge H \wedge G, S_h, \mathbb{S}, \mathbb{B})$

$$= \left\{ \theta \mid H = \theta(S_h) \wedge \mathcal{D}_{\mathcal{B}} \models \mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge G \wedge \text{agg_cond}(A, S_h \cup \mathbb{S}, \mathbb{B})) \right\}$$

where H and S_h are conjunctions of CHR constraints, G and \mathbb{B} conjunctions of built-in constraints, A is a conjunction of aggregates, and \mathbb{S} is a set of identified CHR constraints (a CHR store).

Definition 4 (Aggregate Condition). For an aggregate A of the form $\text{aggregate}(s, i, d, f, T, G, R)$, a CHR store \mathbb{S} and a built-in store \mathbb{B} :

$$\text{agg_cond}(A, \mathbb{S}, \mathbb{B}) = s(V_0) \wedge \bigwedge_{k=1}^n i(V_{k-1}, \theta_k(T), V_k) \wedge f(V_n, R)$$

where V_0, \dots, V_n are new variables and $\{\theta_1, \dots, \theta_n\} = \bigcup_{H \subseteq \mathbb{S}} \text{matchings}'(G, H, \mathbb{S}, \mathbb{B})$. The condition is extended to conjunctions of aggregates in the obvious manner.

In its generic syntactic form (cf. Section 2.1), the head of a rule is prepended with a conjunction of aggregates A (recall that an aggregate's location in the head has no semantic meaning):

Definition 5 (Transition rules). Given a CHR program \mathcal{P} , execution proceeds by exhaustively applying the following transition rules, starting from an initial state of the form $\langle \mathbb{G}, \emptyset, \text{true}, \emptyset \rangle_1$:

1. **Solve.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where c is a built-in constraint and $\mathcal{D}_{\mathcal{B}} \models \exists_{\emptyset} \mathbb{B}$.
2. **Introduce.** $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle \mathbb{G}, \{c\}\#n \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c is a CHR constraint and $\mathcal{D}_{\mathcal{B}} \models \exists_{\emptyset} \mathbb{B}$.

3. **Apply.** $\langle G, H_1 \cup H_2 \cup S, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle B \uplus G, H_1 \cup S, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
 where $\mathcal{D}_{\mathcal{B}} \models \exists_0 \mathbb{B}$ and \mathcal{P} contains a rule $r @ A, H'_1 \setminus H'_2 \iff G \mid B$ and
 $\theta \in \text{matchings}((A, H'_1, H'_2, G), H_1 \cup H_2, S, \mathbb{B})$ and $h = (r, \text{id}(H_1, H_2)) \notin \mathbb{T}$

The propagation history does not record an aggregate in any way, so a rule is never fired more than once with the same combination of constraints, even if the aggregate’s value changes. We call this *fire-once semantics*. More information regarding this choice can be found in [9].

4 Implementation through Program Transformation

The transformation schemes presented here improve earlier schemes described in [9]. Two different aggregate computation strategies are supported: *on-demand* (Section 4.2), and *incremental* (Section 4.3). The source-to-source transformations are implemented in the K.U.Leuven CHR system [6] in SWI-Prolog [12], but the approach is equally applicable to other systems implementing the refined operational semantics. The implementation is based on high-level *meta CHR rules*. Their basic syntax and semantics is outlined first in Section 4.1.

4.1 Meta CHR Rules

Meta CHR rules allow concise specification of CHR source-to-source transformations. They somewhat resemble ordinary CHR rules, both syntactically and semantically. Only, instead of rewriting constraint multisets, they rewrite the CHR rules of another CHR program, called the *object program*.

A meta rule is applicable if its head can be matched with occurrences in a single *object rule*. When a meta rule fires, the occurrences that matched its removed meta occurrences, are removed from the object rule. In a meta rule’s body, the ‘+’ prefix operator adds kept occurrences to the object rule, ‘-’ adds removed occurrences, and ‘?’ adds extra conjuncts to the object rule’s guard. Writing a CHR rule in the body of a meta rule adds this rule to the object program. The `remaining_head/1` operation returns those occurrences of the object rule not matched by the meta rule, and `guard/1` returns the object rule’s guard.

4.2 On-demand Aggregate Computation

This section gradually introduces and explains a transformation scheme for *on-demand aggregate computation*. The scheme, depicted in Figure 1, outputs code in which aggregates are computed from scratch each time they are required.

Lines 1 to 4. The simplification rule removes each occurrence of `aggregate/7`, and replaces it with a guard (line 4). This guard calls an auxiliary CHR constraint¹, `aggi/2`, that computes the aggregate’s result `A`. The `shared_vars/3`

¹ Even though not actually allowed by the CHR language (cf. Section 2), several CHR implementations do support CHR constraints in guards this way, a feature often exploited by expert users. To properly support such guards though, a number of changes were required to the K.U.Leuven CHR compiler ([9] provides an overview).

```

1 aggregate(Start,Inc,_,Final,T,G,A) <=>
2   new_unique_identifier(i),
3   remaining_head(Head), shared_vars(Head, (T-G), V),
4   ?aggi(V,A),
5   +G#on_active, +G#on_removal,
6   ( aggi(V,A) <=> Start(I), resulti(I), matchi(V), geti(R), Final(R,A) ),
7   ( matchi(V), G ==> incri(T) ),
8   ( incri(T), resulti(R1) <=> Inc(R1, T, R2), resulti(R2) ),
9   ( resulti(R), matchi(_), geti(Q) <=> Q = R ).

```

Fig. 1. The core of a transformation scheme using on-demand aggregate computation. For compactness, pseudo code is used. The function of each line is explained below.

predicate returns the variables shared by its first two arguments. It is used to compute V , the list of all variables required to compute the aggregate (line 3–4). The implementation of the aggregate computation (lines 6–9) is discussed below. The identifier i (line 2) ensures all auxiliary functors, such as $\text{agg}_i/2$, are unique.

Line 5 (on_active and on_removal heads). Under the refined operational semantics, by default, the guard added on line 4 is called each time a matching is found for the remaining occurrences. In general, this does not suffice: the aggregate also has to be (re)computed when its outcome changes. To indicate such extra conditions under which a rule, and thus its guard, have to be (re)considered, we introduced two special types of heads: **on_active** heads and **on_removal** heads. An *on_active head* indicates an additional trigger to fire the rule: when constraints matching the **on_active** head are *activated* (i.e. newly added or reactivated, cf. Section 2.2), the rule is tried. Similarly, an *on_removal head* indicates that the rule additionally has to be tried when constraints matching the **on_removal** head are *removed* from the constraint store. Neither of these types of heads is considered when an occurrence in the regular head is active. Both new types of heads are implemented with a straightforward source-to-source transformation. More information can be found in [9].

Line 5 adds the aggregate’s goal G to the original object rule, both as an **on_active** and as a **on_removal** head. This ensures the guard computing the aggregate is called, not only when the remainder of the original head is matched, but also when constraints matching G are added, reactivated, or removed.

Example 4. The rule from Example 2 (Section 3.2) becomes:

```

account(_,C,#on_active, account(_,C,#on_removal,
client(C) ==> agg0(C,Sum), Sum ≥ 25000 | platinum(C).

```

A client’s accumulated balance is thus also (re)computed when the accumulated balance changes, i.e. each time an **account/3** constraint is added or removed.

Issue 1: Updates. Recall the following rule from Example 1 (Section 3.1):

```

deposit(A,X), account(A,C,B) <=> account(A,C,B+X).

```

The above rule is an instance of a common CHR programming pattern, called an *update*: a constraint is removed and immediately replaced with a similar, updated version. In the context of aggregates however, the removal of the former may cause aggregates to be recomputed prior to the insertion of the updated version². This behavior is not always desired. For instance, in the intermediate state right after the above rule removes ‘`account(A,C,B)`’, the accumulated balance in Example 4 would clearly be incorrect.

As a solution, we introduce pragma `passive_removal`. If a constraint annotated with `passive_removal` is removed, no `on_removal` heads are activated:

```
deposit(A,X), account(A,C,B) # passive_removal <=> account(A,C,B+X).
```

Consequently, the aggregate is only recomputed when the new, updated account is added. This allows the CHR programmer to easily specify the desired behavior.

Lines 6–9. The rules performing the actual aggregate computation are added to the object program by lines 6 to 9. Line 6 implements the `aggi/2` operation. First, the intermediate aggregate result is initialized using the aggregate’s `Start` operation. This intermediate result is stored as a `resulti/1` constraint. Then the `matchi/1` constraint is called, causing the intermediate result to be incremented for each matching aggregate goal `G` (lines 7–8). To perform the matching with the aggregate goal `G` (line 7), the variables `V` it shares with the remaining head of the original object rule are needed (line 3).

Example 5. For the `sum/3` aggregate in Example 2 the following code is generated (recall from Example 3 that `sum(T,G,A) ≡ aggregate(=(0),plus,minus,=,T,G,A)`):

```
agg0(C,Sum) <=> 0=I, result0(I), match0(C), get0(R), R=A.
match0(C), account(_,C,B) ==> incr0(B).
incr0(B), result0(R1) <=> plus(R1, B, R2), result0(R2).
result0(R), match0(_), get0(Q) <=> Q = R.
```

If the `sum` aggregate (the accumulated balance) has to be computed, the result is initialized to zero, stored as a constraint, and then incremented with the balance `B` of each matching `account/3` constraint. To perform this match, the variable `C` (the client’s identifier) is indeed required.

The intermediate result `resulti/1` is incremented through the auxiliary constraint `incri/1` (line 8). This way, the propagation history of the rule on line 7 ensures that each matching goal `G` contributes only once. The argument passed to `incri/1` (line 7), and subsequently to `Inc` (line 8), is the aggregate’s template `T`. The refined operational semantics (cf. Section 2.2) ensures that the call to `matchi/1` only returns to the rule body on line 6 after all matchings and increments are performed. A last auxiliary constraint, `geti/1`, is then used to retrieve and remove the computed result (line 8). Finally, this result is finalized using `Final` to obtain the aggregate result `A` (line 6).

² Similar issues were outlined in [11] in the context of negation as absence.

4.3 Incremental Aggregate Computation

The performance of on-demand aggregate computation, described in the previous section, is not always adequate. Aggregates ranging over large portions of the constraint store may be recomputed from scratch many times. In such cases, it is obviously more efficient to maintain the aggregate value incrementally.

```

1 aggregate(Start,Inc,Dec,Final,T,G,A) <=>
2   new_unique_identifier(i),
3   guard(Guard), remaining_head(Head), shared_vars(Head, (T-G), V),
4   +matchi(V,I), +resulti(I,R), ?Final(R,A),
5   ( Head ==> Guard | initi(V) ),
6   ( matchi(V,_) \ initi(V) <=> true ),
7   ( initi(V) <=> Start(R), matchi(V,I), resulti(I,R) ),
8   ( matchi(V,I), G ==> incri(I,T) ),
9   ( incri(I,T), resulti(I,R1) <=> Inc(R1, T, R2), resulti(I,R2) ),
10  ( matchi(V,I)#passive, G#on_removal ==> decri(I,T) pragma no_history),
11  ( decri(I,T), resulti(I,R1) <=> Dec(R1, T, R2), resulti(I,R2) ).

```

Fig. 2. Transformation scheme for maintained aggregates (a basic, first attempt).

The meta rule in Figure 2 illustrates a basic transformation scheme for incrementally maintained aggregates. The scheme is not yet fully correct with respect to the ω_a operational semantics (cf. Section 3.3) though. Subsequent subsections will refine it to deal with certain semantical issues, and more complex aggregates such as nested and non-ground aggregates.

Basic scheme. Similar to the transformation scheme of Section 4.2, aggregate results are stored in `resulti/2` constraints, and `matchi/2` constraints are used to find matches with the aggregate’s goal `G` (lines 8 and 10). The need for the extra argument, an aggregate identifier, is explained below.

Line 4 The aggregate is no longer replaced by a guard that computes the aggregate result, but instead with a `matchi/2` and a `resulti/2` occurrence in the object rule’s head. Both new occurrences are kept because the computed aggregate result may be needed more than once. Line 4 also adds a guard to finalizes the aggregate result.

Incremental maintenance (lines 8–11) The `resulti/2` and `matchi/2` constraints remain in the store, and the rules added by lines 8–11 ensure these results remain consistent. Maintained results are incremented each time a new matching is found for `G` (lines 8–9), and decremented each time such a matching is removed (lines 10–11). For the latter, the `Dec` argument of `aggregate/7` is used. This argument indicates the inverse operation of `Inc`.

Line 10 The different pragmas and annotations in the rule on line 10 warrant extra clarification. The rule must not fire when a `matchi/2` is active, only when constraints matching `G` are removed. Therefore, the `matchi/2` occurrence is made

passive (`pass` is short for `passive`, a common CHR pragma). Reacting to constraint removals is done, as in Section 4.2, using an `on_removal` head. Finally, pragma `no_history` is added, indicating no propagation history has to be kept for this rule. Otherwise, the rule would only fire once per `matchi/2` constraint, as the `on_removal` head is not included in propagation history tuples.

Aggregate identifiers More than one result may have to be maintained at the same time. To ensure the right result is updated after a match is found (lines 8 and 10), we let corresponding `matchi/2` and `resulti/2` constraints share a unique identifier, and pass this to the `incri/2` or `decri/2` constraint. Other than that, the pattern used to increment and decrement the maintained results is the same as the pattern used in Section 4.2.

Initialization (lines 5–7) Eagerly maintaining all possible aggregate results would be overly expensive. Aggregate maintenance is instead only started once a matching is found for the remainder of the head, as realized by the rule added on line 5. The head and guard of this rule are copied from the original object rule (without copying the aggregate head itself), and its body calls an `initi/2` auxiliary constraint. This constraint is removed by the rule on line 6 if the same aggregate result is already being maintained; in the other case, the rule on line 7 initializes a new result, stores it as a `resulti/2`, and adds a `matchingi/2` constraint.

Issue 1: *Multiple Removals.* The basic scheme does not fully implement the ω_a semantics defined in Section 3.3. This subsection addresses a first issue:

Example 6. Consider the following artificial example:

```
a, count(c, Cs) <=> Cs \== 2 | writeln(Cs).
b \ c, c <=> true.
```

where the “`count(c, Cs)`” aggregate counts the number of `c/0` constraints. Now consider the query “`c, c, a, b`”. First two `c` constraints are added, then `a` is called. The latter causes the `count/2` aggregate to be computed. As the result is equal to two, the first rule does not fire. After adding `b`, the second rule fires and removes both `c` constraints. Suppose the count is maintained incrementally. The removal of the first `c` constraint causes the maintained result to be decremented. The count becomes equal to one, causing the first rule to fire with `Cs` equal to *one*, even though there are *no c constraints left*. This is clearly not correct. The reason is that, whilst both `c` constraints are removed simultaneously, the updates to the maintained aggregate are performed, and visible, one by one.

Our solution is based on splitting the activation of `on_removal` heads into two phases: `on_removal1` and `on_removal2`. When a rule fires, the removed constraints are first matched against `on_removal1` heads. Only after this is done for *all* removed constraints, the same is repeated for the `on_removal2` heads.

Lines 10–11 of Figure 2 are replaced with those in Figure 3. The rules added on lines 10*–11* ensure that first, in the `on_removal1` phase, all affected aggregates are made consistent. The updated results are not yet used immediately as in Example 6. Instead, the `resulti/2` constraint is added *passively* to the

```

...
10* ( matchi(V,I)#passive, G#on_removal1 ==> decri(I,T) pragma no_history),
11* ( decri(I,T), resulti(I,R1) <=> Dec(R1,T,R2), resulti(I,R2)#passive)
12* ( matchi(V,I)#passive, G#on_removal2,
13*   resulti(I,_)#Id ==> chr_reactivatei(Id) pragma no_history).

```

Fig. 3. Code to replace lines 10–11 of the transformation scheme of Figure 2 to correctly deal with multiple constraint removals. Several new lower-level CHR constructs are used. Their semantics is explained in the accompanying text.

constraint store, that is, without searching for matching occurrences. Hence the ‘#passive’ annotation in the body of the rule on line 11*. The results only become active once *all* results are guaranteed consistent again, that is, in the `on_removal2` phase (line 12*). Activating a constraint is done using the low-level `chr_reactivate/1` primitive (line 13*).

Issue 2: Updates. The update pattern causes similar issues in the context of incrementally computed aggregates as described before in Section 4.2, Issue 1. The solution is analogous as well, only with a slightly refined semantics of `pragma passive_removal`: if a constraint annotated with `passive_removal` is removed, no `on_removal2` heads are activated, only `on_removal1` heads (cf. previous issue). As such, the maintained result is passively decremented, but the aggregate only becomes active when the new, updated account is added.

Issue 3: Nested Aggregates. A second semantical problem with the basic transformation scheme occurs when applying it to nested aggregates. The maintained value of a nested aggregate is incremented and decremented using the update pattern. The outer aggregate consequently observes the intermediate state in which the `result/2` constraint holding the old maintained value of the nested aggregate is removed and the new, updated version is not yet added. The solution consists of slightly adjusting lines 9 and 11 in Figure 2, and line 11* in Figure 3, to use `pragma passive_removal` for updates to `result/2` constraints.

Issue 4: Propagation Histories. The transformation scheme adds two extra heads per aggregate. However, according to the ω_a semantics these are not allowed to be part of the propagation history. `Pragma history/2`, introduced in [9], can be used to explicitly specify which occurrence identifiers have to be included in the history tuples. Thus the issue is solved by adding the following code after line 3 of Figure 2 (`histi` is a unique identifier):

```
..., identifiers(Head, Ids), pragma( history(histi, Ids) ), ...
```

Issue 5: Non-ground Aggregates. Two final issues occur when aggregating over goals containing non-ground variables:

- A single built-in constraint (e.g. unification) may cause *multiple goals* to match. The problem is analogous to Issue 1 on *multiple removals*. It has to be ensured that *first* all aggregates are updated, i.e. incremented in this case, *before* the aggregate results are activated.

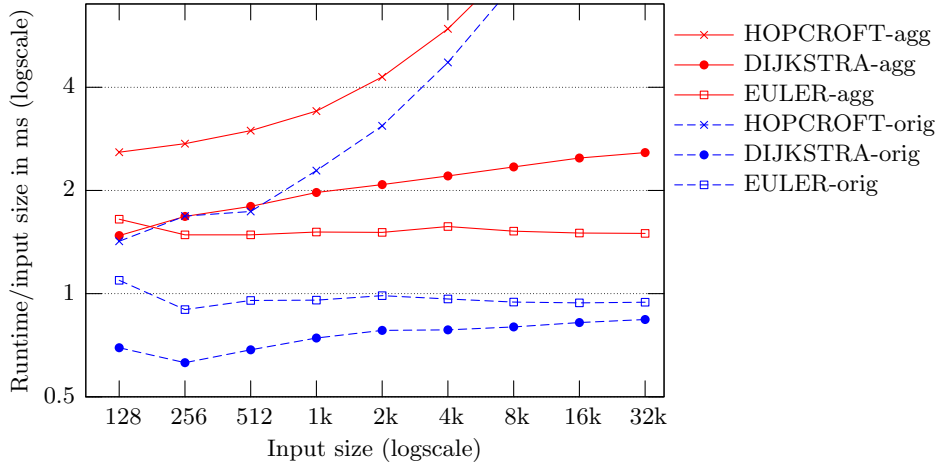


Fig. 4. Runtimes for three programs, with and without aggregates.

- A unification can cause two or more `match/2` constraints to coincide. To preserve correctness, we would have to add the following rule to Figure 2:

$$\dots, (\text{match}_i(V, _) \setminus \text{match}_i(V, I), \text{result}(I, _) \Leftrightarrow \text{true}), \dots$$

Unfortunately, the refined operational semantics (which is used to execute the result of the transformation), does not determine the order in which constraints are reactivated (cf. Section 2.2). This implies there is no clear-cut way to ensure all aggregates are made consistent, or duplicate maintained results are removed, *before* other CHR constraints are reactivated and use the incorrect aggregate values. This lack of control is a general problem of current CHR systems, that warrants further research outside the scope of this paper (see also [2] and Section 5). Fortunately, most aggregates range over ground data. For aggregates ranging over non-ground data, only the on-demand transformation is correct.

5 Discussion and Evaluation

Performance Evaluation. In [8, 9] we revised a number of existing CHR programs to use aggregates. Because our transformation schemes have to deal with all possible use patterns of aggregates, and the original programs are manually specialized, we expect the programs using aggregates to be slower than the original programs. Our prototype implementation however shows the runtime complexity can be maintained, with an acceptable constant overhead. Figure 4 plots benchmark results for the different versions of the DIJKSTRA, EULER, and HOPCROFT programs (cf. [8, 9]). The DIJKSTRA-agg program is about three times slower than the manually specialized DIJKSTRA-orig. For EULER and HOPCROFT, the version with aggregates is only about 1.5 times slower.

The DIJKSTRA-agg program uses an incrementally maintained `min` aggregate. The implementation of this aggregate relies on an efficient priority queue implementation. This illustrates another advantage of language support for aggregates: the data structures required for efficient aggregate computation only have to be implemented once; end users no longer have to worry about this.

For the above figures, a transformation scheme presented in [9] is used for the incrementally maintained aggregates. This scheme is an extended version of the scheme of Section 4.3, in which aggregates are still replaced by guards. The incremental scheme of Section 4.3 considerably improves the latter scheme: it permits efficient indexing on aggregate results, and failing guards no longer backtrack over result maintenance. For the above benchmarks though, we expect no significant difference in performance.

Discussion. Section 4 indicated several issues that occur when transforming to CHR code. A common thread is the lack of control offered by the refined operational semantics, a problem also perceived outside the context of aggregates (cf. [2]). Whilst the low-level constructs we introduced in this paper are acceptable for generated code or expert use, more high-level, declarative control structures are required for the CHR programmer. A first step are the *user-definable rule priorities* introduced by [2].

Related Work. Constructs related to aggregates are found in many languages. For SQL [10], which unlike CHR [7] is not Turing-complete, aggregates do add computational power. The original SQL standard only supports five aggregates: `min`, `max`, `count`, `sum`, and `avg`. Many recent database systems also include the possibility to extend the database query language with user-defined aggregates.

Recently, several production rule systems introduced a general `accumulate` construct, similar to our `aggregate/7`. As far as we know, current versions lack support for nested aggregates, complex goals, and incremental maintenance.

In logic programming, the best-known practical implementation of aggregates are the *all solutions* predicates `findall/3`, `bagof/3` and `setof/3`. Other aggregates can be implemented in terms of these all solutions predicates.

In [11] we introduced $\text{CHR}^{\bar{}}$, an extension of CHR with negation as absence. CHR with aggregates is a far more expressive generalization of $\text{CHR}^{\bar{}}$ as negation as absence can easily be expressed using the `count/2` aggregate.

6 Conclusion and Future Work

In this paper we presented an implementation approach for aggregates, a new declarative language feature for CHR that considerably increases its expressiveness. The approach is based on source-to-source transformation to regular CHR (extended with some low-level constructs). As a side-effect of our work, we created a practical, high-level source-to-source framework based on meta CHR rules. We outlined the design of non-trivial transformation schemes for on-demand and

incremental aggregate computation, and clearly showed the effectiveness of CHR-to-CHR transformations. The source-to-source implementation approach allows for flexible and rapid implementations, easily portable to existing CHR systems. The current generation of optimizing CHR compilers ensure the desired runtime complexity is achieved, with an acceptable constant overhead. We clearly identified the issues that occur when transforming to CHR, and showed how they can be addressed using newly introduced low-level constructs. Several of these constructs have already proven useful outside the context of aggregates (e.g. [2]).

In future work, various ways can be investigated to improve the efficiency of our aggregates implementation. In particular, both specializations on the source level and dedicated support in the CHR compiler can be considered. Even though source-to-source transformation remains effective for aggregates in their full generality, specific cases can e.g. be distinguished where incremental maintenance of aggregates can be embedded directly in the constraint store insertion and removal operations. Also, static and dynamic analyses can be developed to automatically select the aggregate computation strategy (on-demand, incremental, or maybe hybrid strategies).

References

1. The CHR Home Page. <http://www.cs.kuleuven.be/~dtai/projects/CHR/>.
2. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 25–36, Wrocław, Poland, July 2007.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. Logic Programming*, LNCS 3132, Saint-Malo, France, 2004.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Meda, Christina Lopes, Jean-Marc Loingtier, and John Irwing. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*. LNCS 1241, 1997.
6. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *Selected Contributions, First Workshop on Constraint Handling Rules*, May 2004. Home page at <http://www.cs.kuleuven.be/~toms/CHR/>.
7. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules*, pages 3–17, Sitges, Spain, October 2005.
8. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. In *Fourth Workshop on Constraint Handling Rules*, 2007. To appear.
9. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. Technical Report CW481, Dept. Computer Science, K.U.Leuven, 2007.
10. ISO/IEC 9075:2003: Information technology – Database languages – SQL.
11. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In *Third Workshop on Constraint Handling Rules*, pages 125–139, Venice, Italy, 2006.
12. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Home page at <http://www.swi-prolog.org>.