

# Aggregates in Constraint Handling Rules

## Extended Abstract

Jon Sneyers\*, Peter Van Weert\*\*, Tom Schrijvers\*\*\*, and Bart Demeo

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

**Introduction.** Constraint Handling Rules (CHR) [2–4] is a general-purpose programming language based on committed-choice, multi-headed, guarded multiset rewrite rules. 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.

*Example 1.* Suppose the constraints `account(AccountId,ClientId,Balance)` and `client(ClientId)` constitute a simplified representation of the accounts and clients of a bank. Consider the business rule “A client whose accumulated sum of account balances is \$25,000 or more is a platinum client”. A common approach to encode this rule in CHR is the introduction of an auxiliary constraint:

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

However, the explicit maintenance of such an auxiliary constraint is an inherently cross-cutting concern which requires invasive modifications to many rules, spread throughout the entire program, e.g.:

```
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).    □
```

**Aggregates.** We propose an extension of CHR with aggregate expressions in the heads of rules. Aggregates accumulate information over possibly unbounded parts of the constraint store. We provide a wide range of predefined aggregates, including all aggregates commonly found in related paradigms such as database query languages [1] (i.e. `min`, `max`, `sum`, `count` and `avg`) and production rule systems (i.e. `not`, `exists` and `forall`). The complete list of predefined aggregates, together with a number of example uses, can be found in [5].

*Example 2.* The auxiliary constraint in Example 1 can be avoided using `sum`:

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

No further program changes are required to implement the business rule. □

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

\*\* Research Assistant of the fund for Scientific Research - Flanders (FWO-Vlaanderen).

\*\*\* Post-Doctoral Researcher of the fund for Scientific Research - Flanders.

**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 their own *user-defined aggregates*:

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

The `aggregate/7` construct is expressive enough to specify any aggregate. All predefined aggregates are implemented by it. For instance, `sum(T,G,R)` is defined 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`.

**Complex aggregate goals.** We allow arbitrary conjunctions of CHR constraints and guards in the aggregate goal `Goal`: for example, the expression `count((platinum(C),account(_,C,_)), N)` counts the number of accounts owned by platinum clients. We also allow *nested aggregates*, i.e. aggregates inside the goal of another aggregate. For example, the client `C` with the largest total balance `S` is given by `argmax(S, (client(C), sum(B,account(_,C,B),S)))`.

**Implementation.** We have developed a prototype implementation [6] based on a source-to-source transformation to regular CHR (extended with some low-level compiler pragma’s). The implementation allows aggregate computation using either an on-demand or an incremental strategy. Case studies indicate that aggregates can significantly reduce the program size and improve the readability, while the run-time overhead is an acceptable constant factor.

## References

1. ISO/IEC 9075:2003: Information technology – Database languages – SQL.
2. G.J. Duck, P.J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. Logic Programming*, LNCS 3132, pages 90–104, Saint-Malo, France, 2004.
3. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
4. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
5. J. Sneyers, P. Van Weert, and T. Schrijvers. Aggregates in CHR. Submitted to *4th Workshop on Constraint Handling Rules*, 2007.
6. P. Van Weert, J. Sneyers, and B. Demoen. Aggregates for CHR through program transformation. Submitted to *17th Intl. Symposium on Logic-Based Program Synthesis and Transformation*, 2007.