A Proposal for a Next Generation of CHR

Peter Van Weert, Leslie De Koninck, and Jon Sneyers

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

Abstract. This is a proposal for a next generation of CHR called CHR*2*. It combines the best features of language extensions proposed in earlier work and offers a solution to their main drawbacks. We introduce several novel language features, designed to allow the flexible, high-level specification of readable, efficient programs. Moreover, CHR*2* is backwards compatible, such that existing programs can make use of CHR*2*'s new features, but do not need to be changed.

1 Introduction

Constraint Handling Rules (CHR) [1,2,3] is a high-level programming language extension based on guarded, multi-headed, committed-choice multiset rewrite rules. Originally designed for the declarative specification of constraint-based systems, CHR is increasingly used in a wide range of general purpose applications [2]. Several very efficient implementations of CHR exist, embedded in host-languages such as Prolog, Haskell, and Java.

In recent years, several extensions for CHR have been proposed that add powerful language features. Notable examples are negation as absence in $\mathrm{CHR}^{\neg}$ [4], and user-definable rule priorities in $\mathrm{CHR}^{\mathrm{rp}}$ [5,6]. In both cases, the added expressiveness eliminates the need for ad-hoc low-level programming idioms, commonly found in current programs. This leads to cleaner, more concise programs. We are therefore convinced that these language extensions are an important step towards a credible, practical CHR-based programming language.

Nevertheless, current state-of-the-art CHR implementations have not yet incorporated any of these features. This is partly because the experience gained with prototype implementations has shown certain issues with both language extensions [4,6]. In this exploratory paper, we provide a possible basis for CHR*2*, a next generation of CHR. The goals for CHR*2* are:

– to build further on recent advances towards flexible, useful CHR systems
– to combine the advantages of negation as absence and rule priorities, while avoiding their main disadvantages
– to thereby bridge the gap between CHR and production rule systems
– to remain backwards compatible with existing CHR systems where possible

Aside from integrating and further improving earlier proposals, we also introduce several novel language features, such as priority constraints and cardinality annotations. Together, they constitute a powerful, elegant programming language, designed to be the basis for future, more practical CHR systems.

1.1 Constraint Handling Rules

A constraint $c(t_1, \ldots, t_n)$ is an atom with all t_i's host language values (e.g. Herbrand terms in Prolog). There are two types of constraints: built-in constraints, solved by the underlying host language, and user-defined CHR constraints.

There are three types of Constraint Handling Rules: *simplification rules*, *propagation rules* and *simpagation rules*. They have the following form:

$$\begin{array}{llcl}
\textbf{Simplification} & r \mathbin{@} \quad\; H^r & \Longleftrightarrow & G \mid B \\
\textbf{Propagation} & r \mathbin{@} H^k & \Longrightarrow & G \mid B \\
\textbf{Simpagation} & r \mathbin{@} H^k \setminus H^r & \Longleftrightarrow & G \mid B
\end{array}$$

where r is a unique *name*, the *head* consists of non-empty sequences of CHR constraints H^k and H^r, the *guard* G is an optional conjunction of built-in constraints, and the *body* B is a conjunction of CHR and built-in constraints.

Operationally, a multiset of CHR constraints called the *constraint store* is kept. A rule is *applicable* if the store contains constraints matching its head, and its guard is satisfied. If a rule is *fired* (applied), constraints that matched a sequence H^r are removed from the store, and the constraints in the body are added. This high-level semantics is specified formally by the so-called *theoretical operational semantics* (ω_t) of CHR. The *refined operational semantics* (ω_r) [7] is a more low-level description of how most current CHR implementations work. It is considerably more deterministic than ω_t. It establishes that bodies are executed left-to-right, and treats CHR constraints as procedure calls, where each newly added *active* constraint exhaustively searches for possible matching rules in a top-to-bottom order. A detailed description is found in [7].

For more complete, gentler introductions of CHR, its properties and its operational semantics, we refer e.g. to [1,2,3].

1.2 CHR with Negation as Absence

In [4], an extension of CHR with negation as absence called CHR$^{\neg}$ is introduced. In CHR$^{\neg}$, the left-hand side of a rule can contain so-called *negated heads* (preceded by '\\'), which test for the *absence* of constraints. CHR$^{\neg}$ rules also react to constraint *removal*. Negation as absence adds a nice symmetry to the language. Unfortunately, [4] showed that the combination of negation with the refined semantics' sequential, left-to-right processing of constraints, is problematic.

Example 1. The following rules could be part of a banking application. The first rule creates an account for each client that has none; the second handles deposits.

```
client(X) \\ account(X,_) ==> account(X,0).
deposit(X,Amount), account(X,B) <=> account(X,B + Amount).
```

The `deposit`/2 operation should be atomic. However, in the semantics of [4], there exists a state in which the old `account`/2 constraint (i.e. before the deposit) is removed, but the new `account`/2 constraint (the one after the deposit) is not yet added to the store. In CHR$^{\neg}$, the (temporary) removal of an `account`/2 constraint thus causes the first rule to fire if the client has no other accounts.

In general, such inconsistent, intermediate states are unavoidable when extending the refined operational semantics. We refer to [4,8] for more examples.

1.3 CHR with Rule Priorities

CHR$^{\mathrm{rp}}$, CHR with rule priorities, is introduced in [5,6] to deal with the lack of high-level execution control in CHR. Today, many CHR programs rely on the ω_r semantics for correctness or efficiency. This means that the desired execution strategy of these programs is encoded in the program logic. CHR$^{\mathrm{rp}}$ offers a clearer separation of logic and control.

In CHR$^{\mathrm{rp}}$, every rule is annotated with a user-defined priority, a numeric expression that may depend on arguments of the constraints matching the rule's heads. These then imply a (total) order on applicable rule instances that must be respected when executing the program. Explicitly specifying numeric priorities for each rule often is impractical. This is addressed in Section 2.3.

In order to get a sensible operational semantics, CHR$^{\mathrm{rp}}$ rules are executed *atomically*. That is: *all* constraints in the body are resolved before the next applicable rule instance is searched for. However, while the atomic (batch) execution mechanism of CHR$^{\mathrm{rp}}$ is often desirable, there are cases where incremental execution as in the ω_r semantics of CHR is more suitable. A first common case is when CHR interacts with host language statements that are not pure constraints.

Example 2. Consider the following CHR(Prolog) rule:

```
fact(N,F) <=> N₁ is N-1, fact(N₁,F₁), F is F₁ * N.
```

If this rule's body is evaluated atomically, the '`F is F₁ * N`' Prolog statement will raise an error because `F₁` will still be unbound.

Other obvious examples of host language statements that have to be executed in order are those that produce side effects. Moreover, for certain (non-confluent) programs, the order in which CHR constraints are evaluated matters as well. A good example is the union-find algorithm [9], where the order of union and find operations clearly determines the results returned by the find operations. The need for ordering constraints that represent operations or actions, which commonly occurs in general-purpose programs, was also recognized e.g. in [10].

The operational semantics of CHR$^{\mathrm{rp}}$ does not readily allow expressing left-to-right execution. Instead, tedious auxiliary rules and constraints have to be used. We refer to [6] for several worked-out examples.

2 Syntax

2.1 Left-hand Side

In CHR syntax, the different types of *conditions* that determine a rule's applicability — kept occurrences, removed occurrences, and guard conjuncts —

are grouped in separate segments. Consequently, conditions that logically belong together must often be written separately. For larger, multi-headed rules, this hampers both usability and readability, as these restrictions mostly prohibit them from having an intuitive left-to-right reading.

Example 3. Consider the following rule from the RAM program of [11]:

```
prog(L,CJUMP,R,_), mem(R,X) \ pc(L) <=> X ≠ 0 | pc(L+1).
```

The `mem`/2 occurrence is written to the right of the `prog`/4 occurrence because the latter logically determines the memory cell's address `R`. The `pc`/1 occurrence similarly determines the instruction label `L` required for finding matching `prog`/4 constraints. However, because the `pc`/1 occurrence is removed, it must be written to the right of the backslash. Note moreover that the guard on `X` is separated from the variable's occurrence in the head.

The negated heads of CHR$^\neg$ only further deteriorate the situation: the guard becomes further separated from the positive part of the head.

We therefore propose a more flexible syntax for CHR*2*. All applicability conditions, including the conjuncts of the guard, are written on the left-hand side of the rule. Conjuncts have one of the following forms:

1. `+`c or `-`c, with c a CHR constraint, indicating kept and removed heads. Kept heads are preceded by '`+`', removed heads by '`-`';
2. b, with b a built-in constraint, indicating a *guard*;
3. ~(N), with N a conjunction of CHR and built-in constraints, indicating negated heads and their guards. For single-conjunct N, the parentheses may be omitted.

The arrow symbol '`=>`' is used to separate a rule's left- and right-hand side.

Example 4. The rule from Example 3 can now be written as follows:

```
-pc(L), +prog(L,CJUMP,R,_), +mem(R,X), X ≠ 0 => pc(L+1).
```

Example 5. The following excerpt illustrates the use of negation as absence:

```
-get_min(Min), ~c(X) => Min = -1.
-get_min(Min), +c(X), ~(c(Y), Y < X) => Min = X.
```

Rules without left-hand side are supported. Such rules are useful for specifying the constraints that constitute (part of) the initial constraint store. For rules with trivial body `true`, the '`=> true`' may be omitted.

Example 6. The following two rules illustrate idiomatic use of this syntax:

```
=> min(0).
+min(X), -min(Y), X ≤ Y.
```

The new CHR*2* syntax can readily be used alongside traditional CHR syntax. To ease the transition, '`==>`' implies a default '`+`' for all occurrences, and similar defaults apply for simplification and simpagation rules.

Constraint identifiers In most operational semantics and implementations of CHR, each constraint is assigned a unique identifier. We propose the programmer can access these identifiers explicitly, be it as an abstract data type.

Example 7. In the following rule, we retrieve the identifiers of the two heads and impose that the first one is smaller than the second:

```
idempotence @ +leq(X,Y) # K, -leq(X,Y) # R, K < R.
```

This rule guarantees that, if duplicate `leq/2` constraints are found, the more recent instance is always removed, resulting in better termination behavior.

Other supported operations include `kill(`*id*`)`, which removes a constraint from the constraint store, and `alive(`*id*`)`, to test whether the constraint has been removed or not. Having access to identifiers facilitates source-to-source transformations for language extensions. Example transformations that would have benefited greatly from explicit identifiers include [8] and [5].

2.2 Right-hand Side

In most current systems, constraints in the goal and rule bodies are evaluated sequentially, left-to-right, as specified formally in the refined operational semantics. In CHR$^{\text{rp}}$, however, all constraint conjunctions are evaluated in batch [6]. In Sections 1.2–1.3, we clearly argued why neither approach is optimal. While batch evaluation is often preferred after adding negation or priorities, CHR's conventional sequential execution is imperative for a seamless interaction with the host language. Our solution is thus to support both types of conjunctions: batch conjunction, separated by '`&`', and sequential conjunction, separated by '`,`'. Note that this supports backwards compatibility with existing CHR programs.

2.3 Priority Constraints

In CHR$^{\text{rp}}$, the priority assigned to each rule is an expression that evaluates to an integer number, possibly derived from the arguments of the constraints matched by the rule. Firstly, these priority numbers imply a total preorder over applicable rule instances, whereas the programmer mostly wants to enforce only a partial preorder. Secondly, determining a suited priority number for a rule requires global knowledge of numbers already assigned to other rules. Adding new priority numbers may require renumbering. For larger programs this rapidly becomes problematic.

Therefore, we propose that rules are no longer assigned numbers explicitly. Instead, each rule is assigned a *rule descriptor*, an arbitrary term that may contain variables occurring in the remainder of the head. This is an extension of the rule names traditionally used in CHR systems (note though that rule descriptors no longer have to be unique). Next, *priority constraints* are specified over these descriptors. Supported constraints are $=$, $<$, $\leq$, $>$ and $\geq$, where 'larger' means 'higher priority'. The operands of priority constraints are *rule* *patterns*, or sets thereof, that are matched with rule descriptors either statically or dynamically. A program's priority constraints thus imply a partial preorder on rule instances. We introduce priority constraints by example.

Example 8. The following declaration could be used for the LEQ program:

```
priority transitivity < {reflexivity, idempotence, antisymmetry}.
```

It declares that the `transitivity` rule has lower priority than the other rules. For LEQ, this is required for optimal performance and termination behavior.

The set of rule descriptors matching the first operand is always implicitly subtracted from the set matching the second operand. The above declaration is thus equivalent to the more convenient shorthand: '`priority transitivity < _.`' Obviously, specifying multiple '`... < _`' or '`... > _`' constraints in a single CHR*2* program would be inconsistent. Two special priorities, `lowest` and `highest`, can be used instead; for instance: '`priority transitivity = lowest.`'

Example 9. In [6], the following CHR$^{\text{rp}}$ program was introduced:

```
  1 :: source(V) ==> dist(V,0).
  1 :: dist(V,D₁) \ dist(V,D₂) <=> D₁ ≤ D₂ | true.
D+2 :: dist(V,D), edge(V,C,U) ==> dist(U,D+C).
```

It implements Dijkstra's shortest path algorithm using only three rules (recall that in CHR$^{\text{rp}}$ a lower number indicates a higher priority). The `D+2` priority, however, is somewhat artificial. In CHR*2*, this program becomes:

```
init          @ +source(V) => dist(V,0).
keep_shortest @ +dist(V,D₁), -dist(V,D₂), D₁ ≤ D₂.
label(D)      @ +dist(V,D), +edge(V,C,U) => dist(U,D+C).

priority keep_shortest > label(_), label(X) > label(Y) if X < Y.
```

From CHR*2*'s priority constraints the intended priorities are readily apparent. This example further illustrates the declaration of dynamic priorities by including head arguments in the rule descriptors, and the '`if`' construct to declare conditional priority constraints. Added advantage is that if, for instance, the `init` rule would require a priority lower than all `label(_)` instances, this could be declared in CHR*2* as: '`priority label(_) > init`'. Expressing this in CHR$^{\text{rp}}$ is impossible, as no upper bound on the distance `D` is known a priori.

Example 10. For backwards compatibility, or for instance for smaller programs, integer numbers can still be used to specify priorities. It then suffices to add the following declaration: '`priority X > Y if integer(X), integer(Y), X > Y.`'

3 Operational Semantics

In Section 3.2, we define a high-level, abstract operational semantics for CHR*2*, denoted by ω_t^2. It remains close to the conventional ω_t semantics of regular CHR

[7], but incorporates batch processing, priorities, and negation as absence. Next, in Section 3.3, we specify a more deterministic refined operational semantics ω_r^2 for CHR*2*, designed to describe more closely the intended runtime behavior. In both cases, we will discuss the relation with relevant existing operational semantics [6,7]. First, we define a more convenient normal form for CHR*2* programs.

3.1 Rule Normal Form

Each CHR*2* rule can be reduced to the following normal form:

$$r :: p \ @ \ H, G, \sim(N_1, G_1), \ldots, \sim(N_m, G_m) \Rightarrow B_1, \ldots, B_n \qquad (1)$$

with H and N_i conjunctions of CHR constraints, and G and G_i conjunctions of built-in constraints. In the normal of rules with an empty positive head, $H = \texttt{init}$, with $\texttt{init}/0$ a special CHR constraint. For ease of presentation, we assume that the body is a (non-empty) sequential conjunction of batch conjunctions B_j. A straightforward source-to-source transformation is used otherwise. All constraint removals are made explicit as conjuncts of B_1. We further assume that the rule descriptor p, which determines the rule's priority, depends on at most one conjunct of H; see [6] for a transformation to deal with the general case. Because rule descriptors are not unique, each rule in normal form is assigned a unique *rule identifier* r implicitly (required for the propagation history).

3.2 Abstract Operational Semantics

As conventional in CHR, the ω_t^2 semantics is defined as a state transition system.

Definition 1 (Identified constraints). *An* identified CHR constraint $c\#i$ *is a CHR constraint* c *annotated with a unique identifier* i*. We further introduce the function* $\mathsf{chr}(c\#i) = c$*, and extend it to sequences in the obvious manner.*

Definition 2 (State). *In* ω_t^2*, a* state *is a tuple* $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$*, with the* CHR constraint store $\mathbb{S}$*, the* built-in constraint store $\mathbb{B}$*, the* propagation history $\mathbb{T}$*, and the next free identifier* n *defined as usual (see for instance [2,4,6,7]). The* goal $\mathbb{G}$ *of an* ω_t^2 *state is defined as a sequence of elements of form* $G@p$*, with* G *a sequence of batch conjunctions, and* p *a ground rule descriptor that determines the priority at which that part of the goal has to evaluated.*

Given an initial goal G, i.e. a sequence of batches of constraints, the initial state is $\langle [G' \ @ \ \texttt{lowest}], \emptyset, \texttt{true}, \emptyset \rangle_1$. In G', the first batch of G is extended with $\texttt{init}$ (see Section 3.1). Table 1 shows the transitions of the ω_t^2 semantics.

Definition 3 (Applicability condition). *Given constraint stores* $\mathbb{S}$ *and* $\mathbb{B}$*, a rule instance* (r, I) *is* applicable*, denoted* $\mathsf{applicable}((r, I), \mathbb{S}, \mathbb{B})$*, iff a matching substitution* θ *exists for which* $\mathsf{apply}((r, I), \mathbb{S}, \mathbb{B}, \theta)$ *is defined. The latter partial function is defined as* $\mathsf{apply}((r, I), \mathbb{B}, \mathbb{S}, \theta) = B \ @ \ p$ *iff* $I \subseteq \mathbb{S}$ *and, renamed apart, the rule with identifier* r *is of normal form (1), such that* $\mathsf{chr}(I) = \theta H$ *and* $\mathcal{D} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}} \Big(\theta(G) \wedge \bigwedge_{i=1}^{m} \big(\forall X \Subset (\mathbb{S} \setminus I) : \neg \exists \eta : \mathsf{chr}(X) = (\eta\theta)(N_i) \wedge (\eta\theta)(G_i) \big) \Big)$ *where* $\mathcal{D}$ *is the built-in constraint domain and* η *are matching substitutions.*

1. Batch $\langle [[G \mid Gs]@p \mid \mathbb{G}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_t^2}{\rightarrowtail}_P \langle [Gs@p \mid \mathbb{G}], \mathbb{S}', \mathbb{B}', \mathbb{T}'\rangle_{n'}$ where $\langle \mathbb{S}', \mathbb{B}', n'\rangle = \mathsf{batch}(G, \mathbb{S}, \mathbb{B}, n)$, and $\mathbb{T}' = \{(r, I) \in \mathbb{T} \mid \mathsf{applicable}((r, I), \mathbb{S}', \mathbb{B}')\}$. This processes the next batch of body conjuncts: CHR constraints are added or removed, built-in constraints are solved. This transition applies only if no **Apply** transition can fire a rule instance of priority p' with $p' \succ p$.
2. Pop $\langle [\epsilon \mathbin{@} p \mid \mathbb{G}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_t^2}{\rightarrowtail}_P \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$.
3. Apply $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_t^2}{\rightarrowtail}_P \langle [B \mathbin{@} p \mid \mathbb{G}], \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \uplus \{(r, I)\}\rangle_n$ with θ a matching substitution for which $\mathsf{apply}((r, I), \mathbb{S}, \mathbb{B}, \theta) = B \mathbin{@} p$ with p maximal, that is $\neg\exists r', I', \theta' : \mathsf{apply}((r', I'), \mathbb{S}, \mathbb{B}, \theta') = B' \mathbin{@} p'$ with $p' \succ p$. An **Apply** transition must be followed by a **Batch** or a **Pop** transition (to ensure the timely processing of the first batch of the body). In other words: an **Apply** transition cannot follow another **Apply** transition.

Table 1. Transitions of ω_t^2

The batch function, finally, is recursively defined as follows:

- $\mathsf{batch}(\emptyset, \mathbb{S}, \mathbb{B}, n) = \langle \mathbb{S}, \mathbb{B}, n\rangle$
- $\mathsf{batch}(b \mathbin{\&} G, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, \mathbb{S}, b \wedge \mathbb{B}, n)$ if b is a built-in constraint
- $\mathsf{batch}(c \mathbin{\&} G, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, n+1)$ if c is a CHR constraint
- $\mathsf{batch}(\mathtt{kill}(i) \mathbin{\&} G, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, \mathbb{S}', \mathbb{B}, n)$, with $\mathbb{S}' = \mathbb{S} \setminus \{c\#i\}$ if $\exists c\#i \in \mathbb{S}$, and $\mathbb{S}' = \mathbb{S}$ otherwise

Discussion The ω_t^2 semantics is completely compatible with existing semantics. For regular CHR programs, ω_t^2 reduces to ω_t, and for CHR$^{\mathrm{rp}}$ programs (where all conjunctions are batch conjunctions), ω_t^2 reduces to the ω_p semantics of [6]. The correspondence theorems of [6, Section 3.3.3] can easily be extended to programs that combine batch and sequential conjunction.

3.3 Refined Operational Semantics

A central concept in any refined operational semantics for CHR is the *active constraint* [7]. As in the $\omega_r^{\neg}$ semantics of [4], both added and killed constraints can be active. An active constraint traverses all its occurrences (either positive or negative), searching for applicable rule instances. Of course, the program's priority constraints must be respected. For this purpose, as in the ω_{rp} semantics of [6], priority queues are used.

An ω_r^2 state is of form $\langle \mathbb{A}, \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$. The goal $\mathbb{G}$ of ω_t^2 states is replaced by an *activation stack* $\mathbb{A}$, and a *stack of priority queues* $\mathbb{Q}$. Each priority queue contains newly added or killed constraints that still have to be activated. Each item in a queue is annotated with a ground rule descriptor. The $\mathsf{find\_min}$ operation returns the priority queue item with the highest priority rule descriptor.

The transitions of ω_r^2 are given in Table 2. Given a goal G, the initial state is $\langle [G' \mathbin{@} \texttt{lowest}], [\emptyset], \emptyset, \texttt{true}, \emptyset\rangle_1$, where G' is obtained by adding `init` to the first batch conjunct of G.

1. Activate $\langle \mathbb{A}, [Q\vert\mathbb{Q}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle [c\#i : 1 \mathbin{@} p\vert\mathbb{A}], [Q \setminus \{c\#i \mathbin{@} p\}\vert\mathbb{Q}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if $\mathbb{A} = [G \mathbin{@} p_0\vert\_]$ with G a (possibly empty) list, and $\mathsf{find\_min}(Q) = \pm c\#i \mathbin{@} p$ with $\neg(p \prec p_0)$.
2. Batch $\langle [[G\vert Gs] \mathbin{@} p_0\vert\mathbb{A}], [Q\vert\mathbb{Q}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle [Gs \mathbin{@} p_0\vert\mathbb{A}], [Q'\vert\mathbb{Q}], \mathbb{S}', \mathbb{B}', \mathbb{T}'\rangle_{n'}$ where $\langle Q', \mathbb{S}', \mathbb{B}, n'\rangle = \mathsf{batch}(G, Q, \mathbb{S}, \mathbb{B}, n)$, and $\mathbb{T}' = \{(r, I) \in \mathbb{T} \mid \mathsf{applicable}((r, I), \mathbb{S}', \mathbb{B}')\}$. This processes the next batch of body conjuncts: CHR constraints are added or removed, built-in constraints are solved, and the necessary constraints are scheduled. This transition only applies if the **Activate** transition does not apply.
3. Pop $\langle [\epsilon \mathbin{@} p_0\vert\mathbb{A}], [Q_0, Q_1\vert\mathbb{Q}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle \mathbb{A}, [Q_0 \cup Q_1\vert\mathbb{Q}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$. This transition only applies if $Q_0 = \emptyset$ or $\mathsf{find\_min}(Q_0) = c\#i \mathbin{@} p$ with $\neg(p \succ p_0)$.
4. Apply $\langle \mathbb{A}, \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle [B \mathbin{@} p \mid \mathbb{A}], [\emptyset\vert Q], \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \uplus \{I\}\rangle_n$ if $\mathbb{A} = [\pm c\#i : j \mathbin{@} p\vert\_]$, and the j^{th} positive/negative occurrence of c of priority p occurs in rule r (in the positive case, $c\#i$ occurs in sequence S on the position corresponding to this occurrence), with $I = (r, S)$ and θ a matching substitution for which $\mathsf{apply}(I, \mathbb{S}, \mathbb{B}, \theta) = B \mathbin{@} p$.
5. Default $\langle [\pm c\#i : j \mathbin{@} p\vert\mathbb{A}], \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle [\pm c\#i : j{+}1 \mathbin{@} p\vert\mathbb{A}], \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if for every rule instance I for which the **Apply** transition applies in the current state (if any), the following holds: every derivation D starting in the current state has an initial sub-derivation D' in which every state has a priority of at least p, and which ends in a state in which either I fires or I is not applicable. We defined the priority of a state $\langle [\_ \mathbin{@} p'\vert\mathbb{A}], \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ as p'.
6. Drop $\langle [\pm c\#i : j \mathbin{@} p\vert\mathbb{A}], \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \overset{\omega_r^2}{\rightarrowtail}_P \langle \mathbb{A}, \mathbb{Q}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if there is no j^{th} positive/negative occurrence of c of priority p in P.

Table 2. Transitions of ω_r^2

The batch function is extended as follows:

- $\mathsf{batch}(\emptyset, Q, \mathbb{S}, \mathbb{B}, n) = \langle Q, \mathbb{S}, \mathbb{B}, n\rangle$
- $\mathsf{batch}(b \mathbin{\&} G, Q, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, Q' \cup Q, \mathbb{S}, b \wedge \mathbb{B}, n)$
  where b is a built-in constraint and Q' is defined as follows:

$$Q' = \{c\#i \mathbin{@} p \mid c\#i \in S \text{ and } c \text{ has an occurrence of priority } p\}$$

  where the set of *reactivated constraints* $S \subseteq \mathbb{S}$ satisfies the following bounds:
  min: $\forall r \in \mathcal{P}, \forall I \subseteq \mathbb{S} : (\neg\mathsf{applicable}((r, I), \mathbb{S}, \mathbb{B}) \wedge \mathsf{applicable}((r, I), \mathbb{S}, b \wedge \mathbb{B})) \rightarrow (S \cap I \neq \emptyset)$

max: $\forall c \in S : \mathsf{vars}(c) \not\subseteq \mathsf{fixed}(\mathbb{B})$. Furthermore, if $\mathcal{D} \models \mathbb{B} \rightarrow b$ then $S = \emptyset$.

- $\mathsf{batch}(c \,\&\, G, Q, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, Q' \cup Q, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, n+1)$
  where c is a CHR constraint and Q' defined as follows:

$$Q' = \left\{ +c\#n \ @ \ p \mid c \text{ has a positive occurrence of priority } p \right\}$$

- $\mathsf{batch}(\mathtt{kill}(i) \,\&\, G, Q, \mathbb{S}, \mathbb{B}, n) = \mathsf{batch}(G, Q' \cup Q, \mathbb{S} \setminus \{c\#i\}, \mathbb{B}, n)$
  with $Q' = \left\{ -c\#n \ @ \ p \mid c \text{ has a negative occurrence of priority } p \right\}$

Discussion The ω_r^2 semantics is formulated as shown to ensure backwards compatibility with both the (theoretical) priority semantics ω_p of CHR$^{\mathrm{rp}}$ for CHR$^{\mathrm{rp}}$ programs (i.e. no negation, only batch conjunction) and the refined operational semantics of CHR for regular CHR programs (i.e. no negation, no priorities, only sequential conjunction). It based on the refined priority semantics ω_{rp} of CHR$^{\mathrm{rp}}$. Its constraint activation policy is as follows. After processing a batch (i.e. part of the initial goal or a rule body), every CHR constraint in the batch and every CHR constraint affected by a (built-in) constraint in the batch, is activated at each priority higher than or equal to that of the previously active constraint. If no rule body consists of more than one batch, the activation policy is consistent with the priority semantics of CHR$^{\mathrm{rp}}$. Indeed, if after firing a rule instance I, the next instance to be fired is of a higher priority, then the latter can only be applicable because of firing I. Furthermore, the semantics only requires the active constraint to be interrupted by constraints at a (strictly) higher priority in this case. For programs in which all conjunctions are of the sequential type and all rule priorities are equal, the activation policy is the same as that of the ω_r semantics of regular CHR.

The **Default** transition allows any rule instance to be skipped that was already applicable prior to the (re-)activation of the active constraint (if it has not already fired, these rule instances are known to fire in some later state, if they do not become inapplicable before that). While not supported by the refined operational semantics of CHR, certain implementations do implement such optimizations to some extent, as indicated in [12]. It further allows certain optimizations when implementing negation as absence.

4 Annotations

Besides rules, a CHR*2* program also contains constraint declarations, as well as additional annotations stating program properties and invariants. While some invariants can be derived by automatic program analysis, for instance using the abstract interpretation framework of [13], this is not the case in general. Annotations such as those introduced in this section are invaluable as part of a program's documentation, and constitute crucial hints for compiler optimizations. They could also be verified automatically if desired, either when running the program in some debugging mode, or by static program verification.

4.1 Type and Mode Declarations

Most current CHR systems allow for type and mode declarations of constraint arguments. However, practice shows that these declarations lack expressiveness.

Example 11. Consider the following declaration: '`constraint fib(+int, ?int).`' (the syntax is based on that of the K.U.Leuven CHR system). It specifies the first argument of `fib`/2 must always be a ground integer value. The second argument has the default mode, that is: it may be either bound to an integer, or a free variable. However, this does not reflect the following important property: in the final constraint store, the second argument of `fib`/2 constraint will always be bound (to a computed Fibonacci number in this case).

We therefore introduce *temporal modifiers*. For the above example, a more precise declaration is '`constraint fib(+int, {?int in goal, +int in result}).`' The following temporal modifiers are supported:

– '`in goal`': in the initial state
– '`in result`': in the final state
– '`at R`': in states in which rules whose descriptor match pattern `R` may be applicable; that is: all rules of strictly higher priority have been applied
– '`after R`': after all rules with matching descriptor have fired exhaustively

The earlier introduced priorities `highest` and `lowest` can be used as well. The default modifier is '`at highest`'.

4.2 Constraint Invariants

CHR constraints often obey certain invariants such as set semantics or functional dependencies. In CHR, it is common practice to enforce these invariants by adding appropriate rules.

Example 12. An example rule that enforces set semantics for `leq`/2 constraints was seen in Example 7. The following rule declares that the `fib`/2 constraint of Example 11 has set semantics, and that its first argument functionally determines the second: `+fib(N,`M_1`), -fib(N,`M_2`), ground(`M_1`) =>` M_1 `=` M_2`.`

Because set semantics is very common, and because explicitly specifying a rule is cumbersome, we provide syntactic sugar for it. The `set` declaration ensures that whenever two identical constraints are encountered, the most recent one is automatically removed. For example, the following constraint declaration enforces set semantics for the `leq/2` constraint: '`constraint leq/2 :: set.`' It is functionally equivalent to the `idempotence` rule of Example 7. In general, such rules would also have to be declared to have the highest priority.

Unavoidably, checking for duplicates involves a runtime overhead. Often, however, the programmer knows in advance that duplicates will never be added in the goal or by the program. To state this invariant, the `⋆set` is provided. While not affecting the operational semantics, it serves as valuable program documentation, and can be exploited by the compiler in a number of ways.

Similar shorthand declarations for functional dependencies are equally valuable, but can be expressed as well using the cardinality annotations of Section 4.3.

4.3 Cardinality Annotations

A crucial aspect of CHR compilation is efficiently finding applicable rules. Given an active constraint, matching partner constraints have to be searched (cf. Section 3.3). The order in which partner constraints are matched is called the *join order* [14]. Asides from indexing, this order is the single most crucial factor determining the program's runtime complexity. Statically determining the optimal join order is hard, as the compiler has only limited knowledge on constraint cardinalities and guard selectivities (see [14]).

Unlike set semantics and functional dependencies, (asymptotic bounds on) cardinalities of constraints not readily expressible by means of CHR rules. In CHR*2*, we therefore support *cardinality annotations*. They help the compiler select proper index structures, and find better join orders. As part of the program's documentation, and even if the compiler cannot exploit them, they provide a synopsis of the constraint store's structure and its evolution over time.

Basic syntax. General cardinality annotations express properties of the size of multisets of constraint tuples (partial joins). These properties are again expressed using constraints. Supported constraints are $=$, $<$, $\leq$, $>$ and $\geq$. The operands are arithmetic expressions of numeric constants, cardinality expressions, cardinality variables, and asymptotic bound expressions.

Cardinalities A cardinality expression is of the form `#{`*LHS*`}` *T*, where *LHS* is an expression with the same syntax as left-hand sides of rules, with the '`+`' and '`-`' prefixes omitted, and *T* is a temporal modifier as defined in Section 4.1. A cardinality expression denotes the size of the multiset matching *LHS*, i.e. the number of applicable instances of a rule with head *LHS*, at time points given by *T*. If no time expression is given, the default is '`at highest`'.

Cardinality variables Cardinality variables such as `n`, `m`, or `num_cs` can be used as a symbolic placeholder for the cardinality of some set, or for relative comparisons of asymptotic cardinality bounds.

Asymptotic bounds Supported asymptotic bound expressions include: '`o(1)`' (constant), '`o(n)`' (linear), '`o(n^K)`' (polynomial; `K` is a real number), '`o(2^n)`' (exponential). More complex expressions are also possible.

Example 13. Using this syntax, the following cardinality annotation can be formulated: '`cardinality #{node(_)} = o(n), #{edge(_,_)} = o(n^2).`'; or, equivalently: '`cardinality #{edge(_,_)} = o(#{node(_)}^2).`'

The meaning of cardinality constraints containing asymptotic bound expressions is interpreted as follows:

- '$A =$ `o(`X`)`': A grows asymptotically as fast as X; $A \in \Theta(X)$
- '$A \leq$ `o(`X`)`': A is asymptotically bounded above by X; $A \in O(X)$
- '$A \geq$ `o(`X`)`': A is asymptotically bounded below by X; $A \in \Omega(X)$
- '$A <$ `o(`X`)`': A is asymptotically dominated by X; $A \in o(X)$
- '$A >$ `o(`X`)`': A asymptotically dominates X; $A \in \omega(X)$

Average-case information. For many purposes (e.g. join ordering), we are interested in the *average* size of a multiset. We express this by wrapping the expression '`#`{*LHS*} *T*' in an `e`/1 term. This means that the *expected* size of the multiset is considered, instead of the exact size.

Example 14. In an implementation of a heap data structure which maintains the minimum of a set, it may be the case that *usually* there is one `min`/2 constraint, but in some exceptional cases (i.e., when the heap is empty) there is no such constraint. This can be expressed with the following annotation:

```
 cardinality e(#{min(_,_)}) = 0.99
```

Star notation. The special symbol '`*`' can be used in argument positions in the *LHS* expression. It means that the property holds for every fixed value of that argument that actually occurs in that argument position. For example, the annotation '`cardinality #{fib(*,_)} in result = 1`' means that in the final state, the first argument of `c`/2 functionally determines the other argument. Multiple stars can be used; the statement must then hold for every combination of values that occur in those argument positions. For example, '`#{c(*,*)} in goal =< 1`' means that `c`/2 initially has set semantics, while '`#{c(*,*)} in goal = 1`' is a much stronger statement, saying that `c`/2 initially encodes a full cardinal product, i.e. the goal '`c(x,1),c(y,2),c(x,2)`' is invalid because `c(y,1)` is missing.

Syntactic sugar. The expression '*LHS T*' may be used as an abbreviation for '`#`{*LHS*} $T \geq$ `1`'. As a consequence, if *LHS* is negation-free, the non-existence property '`#`{*LHS*} $T =$ `0`' can be written as '`~`(*LHS*) *T*'.

Example 15. Consider again the Dijkstra program of Example 9. For that program, we can for instance give the following cardinality annotations:

```
% exactly one source node must be given by the user
 cardinality #{source(_)} in goal = 1.
% there should be no dist/2 constraints in the goal
 cardinality ~dist(_,_) in goal.
% it is a simple graph (no parallel edges)
 cardinality #{edge(*,*,_)} =< 1.
% we expect a dense graph: if there are n nodes, we have about n^2 edges
 cardinality e(#{edge(_,_,_)}) = o(n^2).
% the number of dist/2 constraints corresponds to the number of nodes
 cardinality #{dist(_,_)} in result = o(n).
% only one distance per node (rule "keep_shortest" eliminates doubles)
 cardinality #{dist(*,_)} after keep_shortest =< 1.
```

5 Conclusion

In this paper, we laid the foundations of CHR*2*, a next generation CHR-based programming language. CHR*2* solves several practical issues with the current state-of-the-art. We incorporated negation as failure and rule priorities, two very expressive language features, and effectively resolved the disadvantages of earlier proposals. Several novel features further contribute to the expressiveness and practical usability of the language.

First, we proposed a more flexible syntax, eliminating syntactical restrictions of previous specifications. In particular, we introduced symbolic priority constraints and showed they are more flexible, and provide a better separation of logic and control than CHR$^{\mathrm{rp}}$'s numeric priority expressions.

Next, we formally specified an intuitive operational semantics for CHR*2*, that effectively combines priorities, negation, and both batch and sequential evaluation. The result truly combines the best of several worlds, and is moreover backwards compatible with existing CHR programs.

Lastly, we specified several improved and novel annotations, necessary for the high-level specification of non-functional program properties. These annotations are invaluable both as program documentation, and for optimizing compilers to efficiently execute programs.

The result is a very powerful, yet elegant programming language. We hope that by combining recent advanced language features into a unifying language, and by resolving their outstanding issues, these features will (finally) find their way into existing and future systems.

5.1 Future work

It must be noted that this is essentially an exploratory paper, providing a possible but promising direction for future CHR systems. While we resolved many issues of previous proposals, there is still a considerable need for future work. The design of many of the novel language features proposed in this paper is still in a preliminary stage. Many of CHR*2*'s features have been implemented successfully in JCHR 2, a successor of our K.U.Leuven JCHR System [15]. However, more research and hands-on practical experience is required.

Also, other interesting extensions should certainly be considered as well, including nested negation, aggregates [16], probabilities [17], and disjunctions [18] combined with flexible search strategies [6]. We have restricted CHR*2* to a core language. Rule priorities and negation as absence are offered as well by for instance most production rule systems. Furthermore, they form a good basis for allowing the formulation of other language extensions by means of local source-to-source transformations (see for instance [16]): the rule priorities support a high-level form of execution control, and negation as absence adds the detection of, and triggering on, the removal of CHR constraints.

Acknowledgements Jon Sneyers and Leslie De Koninck are funded by Ph.D. grants of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Peter Van Weert is a Research Assistant of the fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

References

1. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Logic Programming, Special Issue on Constraint Logic Programming **37**(1–3) (1998) 95–138
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Theory and Practice of Logic Programming (2009) To appear.
3. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (August 2009) To appear.
4. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with negation as absence. [19] 125–140
5. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: 9th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press (2007) 25–36
6. De Koninck, L.: Execution control for Constraint Handling Rules. PhD thesis, K.U.Leuven, Belgium (November 2008)
7. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: ICLP'04: 20th Intl. Conf. on Logic Programming. Volume 3132 of LNCS., Springer (2004) 90–104
8. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: To $\text{CHR}^{\neg}$ or not to $\text{CHR}^{\neg}$: Extending CHR with negation as absence. Technical Report CW 446, K.U.Leuven, Dept. Comp. Sc., Leuven, Belgium (May 2006)
9. Schrijvers, T., Frühwirth, T.: Optimal union-find in Constraint Handling Rules. Theory and Practice of Logic Programming **6**(1–2) (2006) 213–224
10. Lam, E.S., Sulzmann, M.: Towards agent programming in CHR. [19] 17–31
11. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. ACM Trans. Program. Lang. Syst. **31**(2) (2009)
12. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: Extending arbitrary solvers with Constraint Handling Rules. In: PPDP '03: Proc. 5th Intl. Conf. Princ. Pract. Declarative Programming, ACM Press (2003) 79–90
13. Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for Constraint Handling Rules. In: 7th ACM SIGPLAN Symp. on Principles and Practice of Declarative Programming, ACM Press (2005) 218–229
14. De Koninck, L., Sneyers, J.: Join ordering for Constraint Handling Rules. In: Fourth Workshop on Constraint Handling Rules, U.Porto (2007) 107–121
15. Van Weert, P., Schrijvers, T., Demoen, B.: K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In: CHR '05: Proc. Second Workshop on Constraint Handling Rules. (2005) 47–62
16. Van Weert, P., Sneyers, J., Demoen, B.: Aggregates for CHR through program transformation. In: LOPSTR '07: 17th Intl. Symp. Logic-Based Program Synthesis and Transformation, Revised Selected Papers. (2008)
17. Frühwirth, T., Di Pierro, A., Wiklicky, H.: Probabilistic Constraint Handling Rules. In: WFLP '02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers. Volume 76 of ENTCS., Elsevier (2002)
18. Abdennadher, S., Schütz, H.: $\text{CHR}^{\vee}$, a flexible query language. In: FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems. Volume 1495 of LNAI., Springer-Verlag (1998) 1–14
19. Schrijvers, T., Frühwirth, T., eds.: CHR' 06: Proc. Third Workshop on Constraint Handling Rules, K.U.Leuven, Dept. Comp. Sc., tech. report CW 452 (July 2006)

A Example Derivation

In this section, we give an example derivation under ω_r^2. Let there be given the following rules

```
1 :: -c.
2 :: -e.
2 :: +a => (c & d & e), e.
3 :: -b.
4 :: -d.
```

and priority declaration

```
priority X > Y if X < Y.
```

i.e. rule 1 has a higher priority than rules 2, 3 and 4 etc. Figure 1 shows how an initial goal [`a` & `b`] is solved. Since there are no built-in constraints, we leave out the built-in constraint store $\mathbb{B}$ from execution states. For compactness, we also leave out identifiers and the propagation history $\mathbb{T}$. Finally, we note that the activation stack is always of the form $[g_1 \text{ @ } p_1, c_1 : j_1 \text{ @ } p_1, \ldots, g_n \text{ @ } \texttt{lowest}]$ with g_i a goal (sequence) for $1 \leq i \leq n$. We leave out the priorities of the goals, because they can be derived from the active constraints that interleave them.

The `a` and `b` constraints are scheduled at their respective priorities. Next, the `a` constraint is activated at priority 3. It fires a rule whose body is a sequence of two batches: `c`&`d`&`e` and `e`. The first batch is processed, and the three constraints in it are scheduled. Constraint `c` is activated at priority 1 and is subsequently removed. Next, `e` is activated at priority 2, which is also the priority of the rule instance that created `e`, i.e. the rule instance involving `a`. We require that `e` is activated here to ensure compatibility with the refined operational semantics. The same requirement is not imposed on constraints from the last batch of a rule body to allow for more implementation freedom. No more constraints can be activated as `d` can only be activated at priority 4, which is lower than the priority of still-active constraint `a`. Therefore, the next batch is processed, scheduling the second `e` constraint. Since this is the last batch, we do not need to activate `e` at priority 2 (as said before). We continue by looking for a next instance involving `a` at priority 2. Then we consecutively activate and simplify `e`, `b` and `d`.

$$
\begin{array}{rcl}
 & & \langle[[\mathtt{a} \,\&\, \mathtt{b}]], [\epsilon], \emptyset\rangle \\
\mathbf{Batch} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon], [\{\mathtt{a} \ @\ 2, \mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}\}\rangle \\
\mathbf{Apply} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[[\mathtt{c} \,\&\, \mathtt{d} \,\&\, \mathtt{e}, \mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\emptyset, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}\}\rangle \\
\mathbf{Batch} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[[\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{c} \ @\ 1, \mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{c} : 1 \ @\ 1, [\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Apply} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[[\mathtt{kill(c)}], \mathtt{c} : 1 \ @\ 1, [\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\emptyset, \{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Batch} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon, \mathtt{c} : 1 \ @\ 1, [\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\emptyset, \{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Pop} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{c} : 1 \ @\ 1, [\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\left.\begin{array}{r}\mathbf{Default}\\ \mathbf{Drop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[[\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{e} : 1 \ @\ 2, [\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\left.\begin{array}{r}\mathbf{Apply}\\ \mathbf{Batch}\\ \mathbf{Pop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[[\mathtt{e}], \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}\}\rangle \\
\mathbf{Batch} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon, \mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2\}, \{\mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Pop} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{a} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2, \mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\left.\begin{array}{r}\mathbf{Default}\\ \mathbf{Drop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{e} \ @\ 2, \mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{e} : 1 \ @\ 2, \epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}, \mathtt{e}\}\rangle \\
\left.\begin{array}{r}\mathbf{Apply}\\ \mathbf{Batch}\\ \mathbf{Pop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon], [\{\mathtt{d} \ @\ 4, \mathtt{b} \ @\ 3\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{b} : 1 \ @\ 3, \epsilon], [\{\mathtt{d} \ @\ 4\}], \{\mathtt{a}, \mathtt{b}, \mathtt{d}\}\rangle \\
\left.\begin{array}{r}\mathbf{Apply}\\ \mathbf{Batch}\\ \mathbf{Pop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon], [\{\mathtt{d} \ @\ 4\}], \{\mathtt{a}, \mathtt{d}\}\rangle \\
\mathbf{Activate} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\mathtt{d} : 1 \ @\ 4, \epsilon], [\emptyset], \{\mathtt{a}, \mathtt{d}\}\rangle \\
\left.\begin{array}{r}\mathbf{Apply}\\ \mathbf{Batch}\\ \mathbf{Pop}\end{array}\right\} & \overset{\omega_r^2}{\rightarrowtail}_P & \langle[\epsilon], [\emptyset], \{\mathtt{a}\}\rangle
\end{array}
$$

Fig. 1. Example derivation under ω_r^2