Finally, A Comparison Between Constraint Handling Rules and Join-Calculus

Edmund S. L. Lam[1] and Martin Sulzmann[2]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
`lamsoonl@comp.nus.edu.sg`

[2] Programming, Logics and Semantics Group, IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S Denmark
`martin.sulzmann@gmail.com`

Abstract. We provide a comparison between Constraint Handling Rules and Join-Calculus. Constraint Handling Rules is a concurrent constraint programming language originally designed for writing constraint solvers. Join-Calculus is a process calculus designed to provide the programmer with expressive concurrency abstraction. The semantics of both calculi is based on the Chemical Abstract Machine. Hence, we expect that both calculi share some commonalities. Surprisingly, both calculi have thus far been studied independently, yet we believe that a comparison of these two independent fields of study is long overdue. This paper establishes a first bridge between Constraint Handling Rules and Join-Calculus as a basis for future explorations. Specifically, we provide examples showing that Join-Calculus can benefit from guarded constraints and constraint propagation as found in Constraint Handling Rules. We provide a compilation scheme for such an enriched Join-Calculus by applying the constraint matching methods of the refined operational Constraint Handling Rules semantics.

1 Introduction

Constraint Handling Rules (CHR) [8] is a concurrent committed-choice constraint logic programming language to describe rewritings among multi-sets of constraints. Join-Calculus [7] is a process calculus designed to provide expressive concurrency abstractions in the form of reaction rules, known as Join-Patterns. Rule triggering depends on the availability and simultaneous consumption of messages received from various shared channels.

The Chemical Abstract Machine (CHAM) [3] provides the semantic foundations for both calculi. We therefore expect that both calculi share some common features. Surprisingly, CHR and Join-Calculus have been studied so far in complete isolation. We believe that a comparison between both calculi is long overdue and should enable a fruitful exchange of ideas and results. To the best of our knowledge, we are first to conduct such a comparison. In this paper, we make the following contributions:

$\boxed{\text{Primitives:}}$

Process Name	p	Variable	x
Constant Value	v	List of a's	$\overline{a}$

$\boxed{\text{Join-Calculus Essentials Expressions:}}$

$$\begin{array}{rccll}
\text{Term} & t & ::= & x & \mid \quad v \\
\text{Process} & P & ::= & p(\bar{t}) & \\
\text{Concurrent Processes} & M & ::= & P & \mid \quad M\ ,\ M \\
\text{Join-Pattern} & J & ::= & P & \mid \quad J \mid J \\
\text{Join-Body} & B & ::= & P & \mid \quad B \mid B \\
\text{Reaction Rule} & D & ::= & J \rhd B &
\end{array}$$

Fig. 1. Join-Calculus Essentials

- We illustrate programming in CHR and Join-Calculus by example. It is known that both calculi are Turing-complete, hence, equally expressive. However, we show that Join-Calculus can benefit from having CHR style guarded and propagated constraints (Section 2.3).
- We introduce a new compilation scheme for Join-Patterns, which is essentially based on the CHR rule matching semantics (Section 3). This allows us to straight-forwardly introduce CHR features like guards, propagation and shared variables (non-linear patterns) into the Join-Pattern world.
- We investigate the commonalities and differences among the standard compilation schemes for rule matching of CHR rewrite rules and Join-Calculus reaction rules (Section 3.3).

Section 2 introduces Join-Calculus informally. We assume that the reader has some basic knowledge of CHR. We conclude and discuss future works in Section 4.

This paper is a revised and extended version of [14]. Our focus here is on a more detailed comparison among Join-Calculus and CHR. The issue of how to integrate CHR into a host language such as Haskell is left for future work.

2 Programming Examples

In this section, we informally introduce Join-Calculus via a simple example: a printer spooler coordinating a network of printers and clients which submits print jobs.

2.1 Join Calculus

Join-Calculus [7] is a process calculus that introduces an expressive high-level concurrency model, aimed at providing a simple and intuitive way to coordinate concurrent processes via reaction rules known as Join-Patterns.

Figure 1 shows the essential core Join-Calculus language. Processes are typically modeled as unique names p each with a fixed number of term arguments. A collection of concurrently running processes (denoted M) is represented by processes composed together with a binary operator ",". This collection is treated as an unordered set of processes. For instance, the following illustrates a collection of concurrent processes, representing the state of the printer spooler, denoted $\mathcal{S}$:

$$\mathcal{S} \quad = \quad \texttt{ready(p1)}, \texttt{ready(p2)}, \texttt{job(j1)}, \texttt{job(j2)}, \texttt{job(j3)}$$

A printer `Pm` which is available for printing will call the process `ready(Pm)`, while a print job `Jn` is submitted to the spooler via calling the process `job(Jn)`. We shall use standard CHR/Prolog notation to distinguish values and variables: Lowercase references for function/constant names and uppercase references for variables. Hence the above illustrates a state consisting of two available printers and three outstanding print jobs. A print job `Jn` is to be matched with any available printer `Pm`, during which printing can be initiated by sending `Jn` to `Pm` (`send(Pm,Jn)`). This behavior is captured by the reaction rule $\mathcal{D}$, defined as follows:

$$\mathcal{D} \quad = \quad \texttt{ready(Pm)} \mid \texttt{job(Jn)} \rhd \texttt{send(Pm,Jn)}$$

A reaction rule $(J \rhd B)$ has two parts. We refer to the left-hand side J as the Join-Pattern and to the right-hand side B as the Join-Body (in our simplified setting rule processes). The Join-Pattern J specifies that processes matching Join-Pattern J can be consumed and replaced by rule processes B. Note that we will sometimes refer to the reaction rules as Join-Patterns as well if there is no ambiguity doing so. A set of reaction rules can be applied to a collection of concurrent processes. This is defined by two forms of transition steps, namely structural steps $(R \vdash M) \rightleftharpoons (R \vdash M')$ and reduction steps $(R \vdash M) \longrightarrow (R \vdash M')$ where R is the set of reaction rules and M, M' are collections of concurrent processes. This exploits the analogy that concurrent processes are a "chemical soup" of atoms and molecules, while reaction rules define chemical reactions in this chemical soup. Structural steps heat/cool atoms to and from molecules (switching to-and-fro ',' and '|'), while reduction steps apply reaction rules to the matching molecules. The following shows a possible sequence of structural/reduction steps which results from applying the printer spooler rule $\mathcal{D}$ on the spooler state $\mathcal{S}$:

$$\mathcal{D} = \texttt{ready(Pm)} \mid \texttt{job(Jn)} \rhd \texttt{send(Pm,Jn)}$$

$$\begin{array}{rl}
 & (\{\mathcal{D}\} \vdash \texttt{ready(p1)}\ ,\ \texttt{ready(p2)}\ ,\ \texttt{job(j1)}\ ,\ \texttt{job(j2)}\ ,\ \texttt{job(j3)}) \\
\rightleftharpoons & (\{\mathcal{D}\} \vdash \texttt{ready(p2)}\ ,\ \texttt{job(j2)}\ ,\ \texttt{job(j3)}\ ,\ \texttt{ready(p1)} \mid \texttt{job(j1)}) \\
\longrightarrow & (\{\mathcal{D}\} \vdash \texttt{ready(p2)}\ ,\ \texttt{job(j2)}\ ,\ \texttt{job(j3)}\ ,\ \texttt{send(p1,j1)}) \\
\rightleftharpoons & (\{\mathcal{D}\} \vdash \texttt{job(j3)}\ ,\ \texttt{send(p1,j1)}\ ,\ \texttt{ready(p2)} \mid \texttt{job(j2)}) \\
\longrightarrow & (\{\mathcal{D}\} \vdash \texttt{job(j3)}\ ,\ \texttt{send(p1,j1)}\ ,\ \texttt{send(p2,j2)})
\end{array}$$

When concurrent processes J' matches a reaction rule $J \rhd B$ (ie. $J' = \theta(J)$ for some substitution θ) causing the rule to be applied, we say that J' has *triggered* the Join-Pattern J. Note the inherent non-determinism in matching processes with Join-Patterns: any pair of `ready(P)` and `job(J)` can be arbitrarily chosen by a structural step and matched with the Join-Pattern.

There are several implementations of Join-Calculus style concurrency abstractions. The JoCaml system [4] is one such example, which introduces Join-

Patterns into the programming language Caml. For instance, the printer spooler can be implemented in JoCaml as follows (omitting details of the function `send`):

```
let ready(P) & job(J) = send(P,J)
in ready(p1) & ready(p2) & job(j1) & job(j2) & job(j3)
```

As shown above, Join-Patterns in JoCaml are declared by the 'let' definition. There are some minor syntax differences ('$\triangleright$' reaction rule symbol is replaced by the more common '=') but it's intended meaning corresponds to it's Join-Calculus counterpart. Also, the symbol '&' replaces both '|' and ',' as a parallel composition operator for specifying both Join-Patterns and concurrent processes. The keyword '`or`' is used to concatenate multiple reaction rules in a 'let' definition. Process calls are treated just like ordinary procedural calls without return values, with the exception that they are matched with Join-Patterns.

On top of Join-Patterns, language extensions like JoCaml also introduces synchronous process calls which returns values, thus provide a synchronization mechanism among concurrent processes. We will omit this to focus our attention on Join-Calculus and CHR.

2.2 Constraint Handling Rules

Independently, Constraint Handling Rules (CHR) [8] has been developed in the field of constraint solving. CHR is a concurrent committed choice constraint programming language. Originally designed for writing constraint solvers, CHRs have through the years been exploited in a wide range of applications [11, 1]. A CHR program essentially consist of a set of multi-headed guarded rules, describing rewritings among multisets of constraints. These rewritings share a striking similarity with Join-Calculus and the chemical abstract machine reductions: CHR rules can be treated as the reaction rules and CHR constraint multiset as the chemical soup.

We illustrate this by reformulating the printer spooler reaction rule from the previous section, with the following CHR program and CHR derivation:

$$\mathcal{P} = \{\texttt{print} \ @ \ \texttt{ready(P),job(J)} \Longleftrightarrow \texttt{send(P,J)}\}$$

$$\begin{array}{ll} & \{\texttt{ready(p1),ready(p2),job(j1),job(j2),job(j3)}\} \\ \rightarrowtail_{\mathcal{P}} & \{\texttt{ready(p2),job(j2),job(j3),send(p1,j1)}\} \\ \rightarrowtail_{\mathcal{P}} & \{\texttt{job(j3),send(p1,j1),send(p2,j2)}\} \end{array}$$

Concurrent processes are represented by the multiset constraint store and CHR derivation steps take the place of the chemical abstract machine reduction steps. Unlike the CHAM semantics which explicitly express matching via structural steps , CHR constraint matching is implicit within derivation steps.

CHR also supports other features like constraint propagation and rule guards which can be possible useful extensions to the Join-Pattern implementations. There are also several well-developed CHR operational semantics [5] and highly competitive state-of-the-art implementations [13, 16].

2.3 Extending Join-Patterns with Guards and Propagation

A particularly useful extension to the Join-Calculus language is Join-Patterns with guard constraints (also known as guarded Join-Patterns), which allows the programmer to express boolean conditions on Join-Patterns. To illustrate guarded Join-Patterns, we consider a more complex variant of the printer spooler: Suppose that there are now conditions which must be met before a print job can be sent to a printer, namely:

– Print jobs have color requirements, namely black-and-white (`bw`), 16 bit color (`color16`) or 32 bit color (`color32`). Hence not all printers can execute a given print job.
– Only print jobs with authorized identity will be entertained.

To handle these requirements, we represent print jobs as `job(J,Cid,Cr)`, where additional arguments `Cid` and `Cr` are the client identifier and color requirements respectively. Available printers also additionally report their capabilities (ie. `ready(P,Cr)`). A new process `auth(Cid)` is also introduced to indicate that `Cid` is an authorized client. Note that we assume that the color values are ordered to reflect their increasing requirement levels (ie. `bw` < `color16` < `color32`). This new printer spooler can be implemented via the following guarded Join-Pattern, expressed in a pseudo JoCaml extension:

```
let auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr)
    when (Cid1 == Cid2) && (Pcr ≥ Jcr) = auth(Cid1) & send(P,J)
in ...
```

We assume that guard constraints are followed by the "`when`" keyword, and `&&` represents logical conjunction. Note that `auth(Cid1)` is "propagated", meaning that it is reintroduced in the Join-Body. This is because `Cid1` should remain authorized even after a print job is submitted.

Existing Join-Pattern implementations (eg. JoCaml [4], Polyphonic C# [2]) use a Join-Pattern compilation scheme [6] that maintains the states of Join-Patterns to determine when they can be triggered during runtime. (we briefly discuss this scheme in Section 3.1). Unfortunately, as indicated in [2], such compilations do not directly support Join-Patterns with guard constraints. It may seem that guarded Join-Patterns can be easily encoded in basic Join-Patterns. For instance, one attempt in JoCaml could be as follows:

```
let auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr) =
      if (Cid1 == Cid2) && (Pcr ≥ Jcr)
      then auth(Cid1) & send(P,J)
      else auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr)
in ...
```

This encoding uses a standard if-then-else conditional statement to replicate the semantics of a guarded Join-Pattern: if conditions are favourable then proceed as normal, otherwise abort by replenishing the involved processes for future matching. While this encoding ensure correctness (authorized print jobs are only sent to printers with sufficient capabilities), it cannot ensure completeness (all authorized print jobs will be submitted to any available and capable printer). This is because standard Join-Pattern compilation schemes do not test all combinations of concurrent processes, as it is unnecessary when matching is merely a test of presence/absence of processes (This is true for standard Join-Patterns). To date, no existing implementations provide efficient and practical compilation of Join-Patterns with guard constraints.

Yet guard constraints are natively supported in the CHR framework. Furthermore, CHR provides other features like propagation and non-linear patterns (variables appearing in multiple unique locations), which would be obviously useful if available in Join-Patterns. For instance, we can represent the new printer spooler Join-Pattern as the following CHR rule:

`print` @ `auth(Cid)` \ `ready(P,Pcr),job(J,Cid,Jcr)` $\Longleftrightarrow$ `Pcr` $\geq$ `Jcr` | `send(P,J)`

CHR rules natively support propagation: constraints like `auth(Cid)` are known as propagated head (appearing before the '\' symbol), which are necessary for triggering the rule but not deleted after its application. Variables are also allowed to appear in multiple locations of the rule head (non-linear patterns). This approach will also benefit from well-developed CHR operational semantics [5] as well as existing CHR optimizations such as hash indexing and optimal join-ordering. Hence we believe that the CHRs would be the ideal solution to handle guarded Join-Patterns, offering a highly optimized and well-studied multiset rewriting operational semantics that shares a similar foundational semantics (chemical abstract machine).

3 Compilation Schemes

In this section, we review the compilation scheme which is currently the de-facto standard for Join-Pattern implementations. Following this, we introduce a new compilation scheme which is based on CHR rule matching.

3.1 Standard Join-Pattern Compilation Scheme

Existing Join-Pattern implementations compile Join-Patterns into state machines that maintain the matching states of the Join-Patterns [6]. For instance, in JoCaml, this compilation involves constructing n message channels (which are typically queues) and a finite state machine (automaton) for each '`let`' definition, that keeps track of the *matching status* of the n queues. Note that '`let`' definitions can contain any finite number of Join-Patterns delimited by the '`and`' keyword. Each message channel is assigned to a unique process name, and represents the collection of calls to this process by concurrent computation threads. Hence, a call to a process is analogous to the arrival of a new message in the corresponding message channel.

States of this automaton are essentially tuples of n bits, one assigned to each message queue stating whether it is empty (`0`) or non-empty (`N`). This automaton is updated every time a new process is call (ie, a new message has arrived) or when a Join-Pattern is successfully matched. Figure 2 shows an example of a `let` definition, consisting of two Join-Patterns, as well as it's corresponding matching status automaton that is constructed. We label the first Join-Pattern as `J1` and the other as `J2` (Note that this labeling is for our presentation only and not part of the semantics).

Each edge labeled with a transition label, which is either of the form `m-j` stating that arrival of message `m` has triggered Join-Pattern `j`, or just `m` which states

Implementation in JoCaml:

```
let p() & q() = a()     (J1)
or p() & r() = b()      (J2)
in ...
```

Chemical Abstract Machine Interpretation:

$(\{\mathtt{J1},\mathtt{J2}\} \vdash ...)$
where $\mathtt{J1} = \mathtt{p}() \mid \mathtt{q}() \triangleright \mathtt{a}()$
$\qquad\mathtt{J2} = \mathtt{p}() \mid \mathtt{r}() \triangleright \mathtt{b}()$

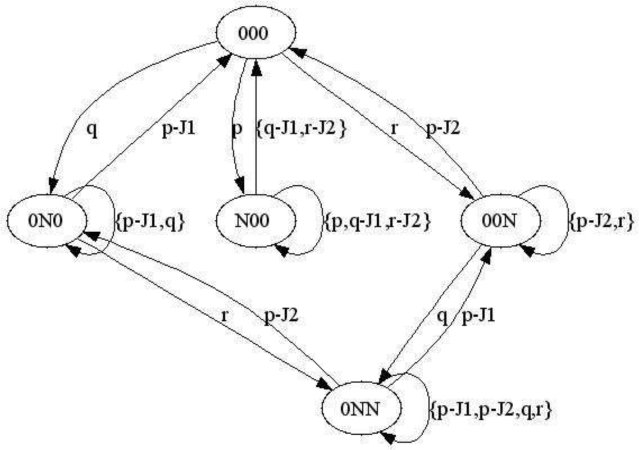


Fig. 2. A Matching Status Automaton for two Join-Patterns

that arrival of message `m` has triggered nothing and therefore is just queued. If there are more than one alternative transitions between two states of the automaton, we will represent them as a single edge with the set of alternative transitions. Since the Join-Patterns considered do not have guard conditions, messages channels are normally implemented with shared queue data structures, and the task of triggering Join-Patterns is as simple as popping messages from the relevant sets of queues. The choice of taking `p-J1` to `000` or `0N0` depends on whether there are any more `q`'s left after `J1` is triggered.

A state automaton compilation like this implements a matching policy referred to as "match as soon as possible", where join patterns are immediately triggered upon the arrival of the final message that complete it's match. For example, suppose that all queues are currently empty (`000`), the arrival of `q` transits the automaton to `0N0` and does not triggering any join patterns since no complete matches are available. The arrival of `p` completes `J1`'s match and triggers it, hence the automaton makes the `p-J1` transition. Note that the "match as soon as possible" matching policy is sound with respect to the chemical abstract machine semantics, but does not allow all its possible non-deterministic behaviors. We will discuss more of this issue in Section 3.3.

3.2 CHR Rule Matching Compilation Scheme

We introduce a new Join-Pattern compilation scheme, based on CHR rule matching semantics. The idea is to compile Join-Patterns into CHR rules, where known CHR operational semantics can be applied to derive Join-Pattern triggering. Our

CHR Primitives:

Constraint Names	c	Variables	x
Constant Values	v	List of a's	$\overline{a}$

CHR Terms and Constraints:

Substitutions	θ	$::=$	$[v_1/x_1, .., v_n/x_n]$
Terms	t	$::=$	$x \quad \mid \quad v$
CHR Constraints	C	$::=$	$c(\overline{t})$
Numbered Constraints	Cn	$::=$	$C\#n$
Occurrence Constraints	Co	$::=$	$C : i$
Active Constraints	A	$::=$	$C\#n : i$

CHR Matching Sets:

Rule Head	H	$::=$	$Co \quad \mid \quad H \wedge H$
Matching Set	$\mathcal{P}$	$::=$	$\{H\} \quad \mid \quad P \uplus P$

CHR Stores and States:

Stores	St	$::=$	$\emptyset \quad \mid \quad \{Cn\} \uplus St$	
Match States	σ	$::=$	$\langle C, St\rangle_n$	(Initial)
		$\mid$	$\langle A, St\rangle_n$	(Intermediate)
		$\mid$	$\langle \theta, R, St\rangle_n$	(Match Success)
		$\mid$	$\langle \epsilon, St\rangle_n$	(Match Fail)

Fig. 3. CHR Rule Matching Essentials

presentation here is inspired by the refined CHR operational semantics [5]. Figure 3 reviews the essential components of the CHR language. The actual CHR framework is much richer than presented here (eg. guards, propagation, etc..). We will omit these features for simplicity, but note that extending this scheme with guards and propagation is straight-forward.

Lists are denoted by $[x \mid xs]$, where x is the first element and xs the tail. The empty list is denoted by $[\ ]$ and $\bar{x}$ is short for a list of x's. Sets are denoted by $\{x_1, ..., x_n\}$ and multi-set union of two sets S_1 and S_2 is denoted by $S_1 \uplus S_2$.

We are particularly interested in the multi-set matching part of CHR executions, hence we only consider CHR rule heads (rule bodies are omitted). Constraints in rule heads are assigned unique occurrence numbers (eg. $C : i$) with respect to there textural order in the program. Rule heads are matched against constraints in the multi-set store. Stored constraints are numbered (eg. $C\#n$) to distinguish duplicate copies. Constraint matching is driven by an active constraint, $C\#n : i$, which matches numbered constraint $C\#n$ with occurrence i. A matching set $\mathcal{P}$ is a set of CHR rule heads. We define two auxiliary functions *cons* and *maxOccurs*, which returns constraints from numbered constraints and returns the maximum occurrence number from a matching set respectively.

$$\boxed{\text{Single-Step Matching Reduction: } \sigma \rightarrow_{\mathcal{P}} \sigma}$$

(Activate) $$\langle C, St\rangle_n \rightarrow_{\mathcal{P}} \langle C\#n : 1, \{C\#n\} \uplus St\rangle_{n+1}$$

(Match) $$\frac{\begin{array}{c} H_1' \wedge C' : j \wedge H_2' \in \mathcal{P} \\ \exists\theta \text{ such that } \quad \theta(C') = C \quad \theta(H_1') = cons(H_1) \quad \theta(H_2') = cons(H_2) \end{array}}{\langle C\#m : j, \{C\#m\} \uplus H_1 \uplus H_2 \uplus St\rangle_n \rightarrow_{\mathcal{P}} \langle \theta, H_1' \wedge C' : j \wedge H_2', St\rangle_n}$$

(Continue) $$\frac{j < maxOccur(\mathcal{P})}{\langle C\#m : j, St\rangle_n \rightarrow_{\mathcal{P}} \langle C\#m : (j+1), St\rangle_n}$$

(Deactivate) $$\frac{j \geq maxOccur(\mathcal{P})}{\langle C\#m : j, St\rangle_n \rightarrow_{\mathcal{P}} \langle \epsilon, St\rangle_n}$$

$$\boxed{\text{Exhaustive Matching Reduction: } \sigma \rightarrow_{\mathcal{P}}^{*} \sigma}$$

(Transitive) $$\frac{\sigma \rightarrow_{\mathcal{P}} \sigma' \quad \sigma' \rightarrow_{\mathcal{P}}^{*} \sigma''}{\sigma \rightarrow_{\mathcal{P}}^{*} \sigma''}$$

(Match-Success) $$\frac{\sigma \rightarrow_{\mathcal{P}} \langle \theta, R, St\rangle_n}{\sigma \rightarrow_{\mathcal{P}}^{*} \langle \theta, R, St\rangle_n}$$

(Match-Fail) $$\frac{\sigma \rightarrow_{\mathcal{P}} \langle \epsilon, St\rangle_n}{\sigma \rightarrow_{\mathcal{P}}^{*} \langle \epsilon, St\rangle_n}$$

Fig. 4. CHR Multi-set Rule Matching Semantics

Figure 4 formally specifies the CHR rule matching semantics. This semantics is defined in terms of reduction steps ($\rightarrow_{\mathcal{P}}$) which maps matching states to matching states. Matching starts from an initial matching state $\langle C, St\rangle_n$. Transition rules are tried in sequence. Rule (Activate) begins the matching procedure by activating constraint C. This involves adding C to the store and assigning it occurrence number 1. The rule (Match) specifies the successful matching of a CHR rule. Active constraint $C\#m : i$ matches with the i^{th} occurrence of $\mathcal{P}$ and matching partners H_1 and H_2 are present in the store. This leads to the match state $\langle \theta, R, St\rangle_n$ where θ is the matching substitution, R is the rule heads that is matched and St is the remaining store after matching constraints are removed. Rule (Continue) moves the search forward by incrementing the occurrence number of the active constraint, while (Deactivate) ends the search when last occurrence is tried and no match is found. Exhaustive reductions $\rightarrow_{\mathcal{P}}$ defines the reduction sequence from a initial match state to a final state (match success/match fail).

Figure 5 illustrates the CHR-Based Join-Pattern matching semantics. We assume a straight forward compilation scheme for Join-Patterns. Namely, processes (messages) are treated as constraints and Join-Patterns are assigned occurrence numbers, depending on the textual order of their appearance. A CHR-Based Join-Pattern matching state ($R \vdash M, St, n$) consist of the reaction rules R, the concurrent messages/processes M (ie. chemical soup), a CHR store St and store identifier n. It essentially maintains the matching status of a set of

$$\boxed{\text{Primitives:}}$$

$$\begin{array}{llll} \text{Process Name} & p & \text{Variable} & x \\ \text{Constant Value} & v & \text{List of } a\text{'s} & \overline{a} \end{array}$$

$$\boxed{\text{Compiled Join-Pattern Expressions:}}$$

$$\begin{array}{rcclcl} \text{Term} & t & ::= & x & \mid & v \\ \text{Process/Constraint} & P & ::= & p(\bar{t}) & & \\ \text{Concurrent Processes} & M & ::= & P & \mid & M\ ,\ M \\ \text{Compiled Join-Pattern} & J & ::= & P : i & \mid & J \wedge J \\ \text{Reaction Rules} & R & ::= & \{J \rhd M\} & \mid & R \uplus R \end{array}$$

$$\boxed{\text{CHR-Based Join-Pattern Matching State:}}$$

$$\begin{array}{rccll} \text{Matching State} & \mathcal{E} & ::= & (R \vdash M, \emptyset, \mathit{1}) & \text{(Initial)} \\ & & \mid & (R \vdash \emptyset, St, n) & \text{(Final)} \\ & & \mid & (R \vdash M, St, n) & \text{(Intermediate)} \end{array}$$

$$\boxed{\text{CHR-Based Reduction Step: } (R \vdash M, St, n) \longrightarrow (R \vdash M, St, n)}$$

$$\text{(Join-Pattern Triggered)} \quad \frac{\begin{array}{c} R = \{H_{\mathit{1}} \rhd B_{\mathit{1}}, ..., H_n \rhd B_n\} \quad \mathcal{P} = \{H_{\mathit{1}}, ..., H_n\} \\ \langle p(\bar{t}), St\rangle_n \rightarrow^*_{\mathcal{P}} \langle \theta, H_i, St'\rangle_{n'} \quad \text{for some } i \in \{\mathit{1}, ..., n\} \end{array}}{(R \vdash \{p(\bar{t})\} \uplus M, St, n) \longrightarrow (R \vdash \theta(B_i) \uplus M, St', n')}$$

$$\text{(Message Stored)} \quad \frac{\begin{array}{c} R = \{H_{\mathit{1}} \rhd B_{\mathit{1}}, ..., H_n \rhd B_n\} \quad \mathcal{P} = \{H_{\mathit{1}}, ..., H_n\} \\ \langle p(\bar{t}), St\rangle_n \rightarrow^*_{\mathcal{P}} \langle \epsilon, St'\rangle_{n'} \end{array}}{(R \vdash \{p(\bar{t})\} \uplus M, St, n) \longrightarrow (R \vdash M, St', n')}$$

Fig. 5. CHR-Based Join-Pattern Matching Semantics

Join-Pattern reaction rules. Hence, it is the runtime structure that replaces the matching state automaton of standard Join-Pattern compilations. For instance, the following shows Join-Patterns (in JoCaml syntax) and its CHR-Based interpretation:

Implementation in JoCaml:	**CHR-Based Matching State:**
`let p() & q() = a()     (J1)`	$(\{$`J1`,`J2`$\} \vdash ...)$
`or p() & r() = b()     (J2)`	where `J1` $=$ `p():1` $\wedge$ `q():2` $\rhd$ `a()`
`in ...`	`J2` $=$ `p():3` $\wedge$ `r():4` $\rhd$ `b()`

Reduction steps $\longrightarrow$ are defined by two transitions (Join-Pattern Triggered) and (Message Stored), both of which specify arbitrary scheduling of a process for CHR matching (Figure 4) and the respective outcomes of the matching. Transition (Join-Pattern Triggered) is taken when scheduled process is successfully matched (according to CHR matching semantics), hence the corresponding

Join-Pattern is triggered. Transition (Message Stored) is taken when scheduled process fails to match, hence the process is stored as a "message" in the CHR store, awaiting for future matching.

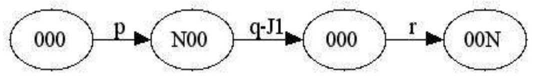
3.3 Discussion

Non-Determinism in CHR Matching Semantics: One important characteristic of this compilation scheme is that multiple CHR matchings are not intended to be done in parallel, but in an interleaving manner (transitions (Join-Pattern Triggered) and (Message Stored) are performed as "atomic" steps, hence cannot be intermediately interleaved). While this would likely limit performance of an actual implementation, it simplifies our CHR matching semantics by a great deal by not allowing execution of multiple active constraints in parallel, thus not introducing more non-determinism than needed. Note that our CHR matching semantics is no more non-deterministic than the refined CHR operational semantics ω_r [5] (Given a fixed sequence of goal constraints to activate, our matching semantics will derive similar results). The main additional source of non-determinism is instead introduced by the top level Join-Pattern reduction steps, which allows processes (constraints) to be scheduled for activation in arbitrary sequences.

"Match as soon as Possible" versus CHR Matching Semantics: The CHR matching semantics shares remarkable similarities with the Join-Pattern state automata "match as soon as possible" matching policy. The Join-Pattern state automata trigger Join-Patterns immediately once a complete match has "arrived", by keeping track on which message channels are empty/non-empty. Similarly, CHR matching semantics executes constraints in sequence of activation and triggers a rule immediately when the CHR store consists of a complete rule head match. Let's consider an example by examining the state transitions taken by the state automaton of Figure 2 in response to the message sequence, `p()`, `q()` then `r()`:

Message Sequence: `[p(),q(),r()]`

Join-Pattern Trigger via State Automata (Figure 2):



This results in the triggering of the join pattern `J1` as it is triggered immediately from the state `N00` once `q()` arrives. This finally leads to a state where only `r()` is left (`00N`). We consider the CHR matching of this example. For brevity, we omit the top-level Join-Pattern reduction steps, but we illustrate the underlying CHR derivations in the sequence of activation: `[p(),q(),r()]`

Join-Pattern Trigger via CHR Matching (Figure 4 and 5):

CHR rule head patterns:
$\mathcal{P} = \{\texttt{J1},\texttt{J2}\}$ where $\texttt{J1} = \texttt{p():1} \wedge \texttt{q():2}$, $\texttt{J2} = \texttt{p():3} \wedge \texttt{r():4}$

Activation Sequence: `[p(),q(),r()]`

$$
\begin{array}{ll}
 & \langle \texttt{p()}, \emptyset \rangle_1 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{p()}\#1 : 1, \{\texttt{p()}\#1\} \rangle_2 \\
\text{(Continue)} \times 4 & \rightarrow_{\mathcal{P}} \langle \texttt{p()}\#1 : 5, \{\texttt{p()}\#1\} \rangle_2 \\
\text{(Deactivate)} & \rightarrow_{\mathcal{P}} \langle \epsilon, \{\texttt{p()}\#1\} \rangle_2 \\
 & \\
 & \langle \texttt{q()}, \{\texttt{p()}\#1\} \rangle_2 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{q()}\#2 : 1, \{\texttt{p()}\#1, \texttt{q()}\#2\} \rangle_3 \\
\text{(Continue)} & \rightarrow_{\mathcal{P}} \langle \texttt{q()}\#2 : 2, \{\texttt{p()}\#1, \texttt{q()}\#2\} \rangle_3 \\
\text{(Match)} & \rightarrow_{\mathcal{P}} \langle [\,], \texttt{J1}, \emptyset \rangle_3 \\
 & \\
 & \langle \texttt{r()}, \emptyset \rangle_3 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{r()}\#1 : 1, \{\texttt{r()}\#1\} \rangle_4 \\
\text{(Continue)} \times 4 & \rightarrow_{\mathcal{P}} \langle \texttt{r()}\#1 : 5, \{\texttt{r()}\#1\} \rangle_4 \\
\text{(Deactivate)} & \rightarrow_{\mathcal{P}} \langle \epsilon, \{\texttt{r()}\#1\} \rangle_4
\end{array}
$$

Similarly, the CHR derivation shows the triggering of rule `J1` and the final CHR store corresponding to `OON` in the state automaton. In both case, Join-Pattern (CHR rule) `J1` is triggered as the match {`p(),q()`} is completed first.

Deleting Matching Transition versus Rule Ordering: CHR rule heads and Join-Patterns may contain overlapping partial matches. Such overlaps are sources of non-deterministic behaviors in the theoretical CHR semantics and Join-Calculus semantics based on the chemical abstract machine. Overlapping partial matches allow multiple rules (CHR rule / reaction rule) to be applicable from certain states, and applying different rules in such states may lead to different outcomes. For example, given Join-Patterns in Figure 2, suppose we are in a state with messages `q()` and `r()`, according to chemical abstract machine, we can trigger either `J1` or `J2` is a `p()` arrives next. This situation is captured by the state `ONN` of the automaton in Figure 2 where we have two edges (`p-J1` and `p-J2`), both triggered by the message `p()`. This indicates that we can trigger either of the two Join-Patterns. In standard Join-Pattern implementations (eg. JoCaml), this is dealt with by allowing the compiler to choose an arbitrary transition edge and remove the other (compiler deletes either `p-J1` or `p-J2`). Hence the final matching status automaton generated would not exhibit such non-determinism.

For the CHR matching semantics, overlapping rules are dealt with in a less ad-hoc way. Consider the CHR matching reductions of this scenario (we assume that CHR store already contains constraints `q()` and `r()` when `p()` is activated:

$$
\begin{array}{ll}
 & \langle \texttt{p()}, \{\texttt{q()}\#1, \texttt{r()}\#2\} \rangle_3 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{p()}\#3 : 1, \{\texttt{q()}\#1, \texttt{r()}\#2, \texttt{p(X)}\#3\} \rangle_4 \\
\text{(Match)} & \rightarrow_{\mathcal{P}} \langle [\,], \texttt{J1}, \{\texttt{r()}\#2\} \rangle_4
\end{array}
$$

Note that the active constraint $\texttt{p()}\#3 : j$ can only match with occurrence j of the CHR program, hence $\texttt{p()}\#3 : 1$ will always try matching with `J1` first before taking transitions (Continue) twice to reach $\texttt{p()}\#3 : 3$ which will try matching with `J2` (Note this never happen since matching with `J1` succeeds). Thus rule textual ordering would prevent such non-deterministic behaviors.

In essence, the CHR refined operational semantics is comparable to a Join-Pattern matching status automaton with a "match as soon as possible" policy and overlapping transitions are deleted depending on textual ordering (ie. transitions involving lower textual ordering join-patterns are deleted). Interestingly, researchers of the two communities proposed very different motivations for enforcing determinism:

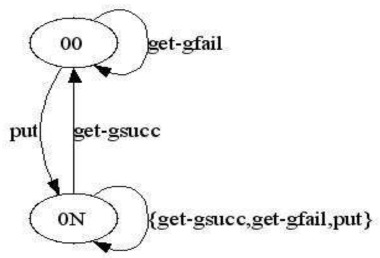
- **Join-Pattern matching status automata:** Non-determinism is removed purely for efficiency (we don't need to decide which overlapping rules to trigger at runtime, because there is at most one), doing so have a price in terms of semantics [6] as some behaviors stated by the theoretical calculus (ie. the chemical abstract machine) can no longer be observable. This suggests that deletion of non-deterministic transitions is viewed as a trade off favouring efficiency over modeling non-deterministic behaviors of concurrent programs.
- **CHR refined operational semantics:** Determinism is motivated not only because of efficiency, but also to make rule-based constraint programming easier (determinism enhances properties like termination and confluence [5]) Rule ordered executions also allow us to write more deterministic programs, which would not work as intended in a unordered setting.

Rule ordered matching semantics introduced by CHR would allow us to write more expressive Join-Patterns. This is illustrated by the following:

```
let get(X) & put(Y) = got(X,Y)      (gsucc)
or get(X) = got(X,ϵ)                (gfail)
in ...
```

The intend is to model a shared buffer, where `get` retrieves an object if one exists, retrieves nothing otherwise, while `put` places an object in the buffer. We assume that ϵ denotes the null object. According to the CHR matching semantics, an active `get(X)` will always try to match in rule order (ie. try `gsucc` before `gfail`). Hence this effectively models the firing of `gfail` in the absence of `put(y)` (ie. get nothing if there are no `put(Y)` found).

The matching status automaton generated by the standard Join-Pattern compilation scheme is illustrated by the following:



Note that arrival of a `get` message at state `0N` will cause two possible transitions (`get-gsucc` or `get-gfail`). Hence, according to standard compilation practice, the compiler will choose either of the two transitions of state `ON` to be deleted from the final automaton. If the `get-gsucc` transitions are removed, only `gfail` will ever be triggered at runtime. If `get-gfail` is removed instead we get the desired behavior for this example. Unfortunately, this choice is not observable to the programmer, hence we cannot make any assumptions on ordering of Join-Patterns in standard Join-Pattern compilations, unlike in our CHR compilation.

Performance versus Expressiveness: Competitive implementations of Join-Patterns (eg. Polyphonic C# [2]) do not explicitly construct state automata but represent Join-Pattern matching states as bitmaps. Hence triggering of Join-Patterns can be executed in constant time, with known bit-masking techniques. The main disadvantage of using such compilation scheme is its incompatibility with the introduction of guard conditions. Our CHR compilation scheme benefits from the straight-forward extensions of features like propagation, guards and non-linear patterns. We also benefit from existing CHR optimizations (eg. constraint indexing, optimal join ordering, passive occurrences, etc.. [5, 9, 12]). Most of such optimizations however, benefit only programs which uses guard conditions and non-linear patterns. Thus, for the class of programs which use only basic Join-Patterns, it is likely that our CHR matching compilation scheme will perform less efficiently compared to the standard schemes.

4 Conclusion and Future Work

We introduced a new Join-Pattern compilation scheme, based on the CHR matching semantics. This matching semantics is inspired by the CHR refined operational semantics. We have shown the basic difference and similarity between the standard Join-Pattern compilation scheme and CHR matching compilation scheme. The main benefits of our CHR matching compilation scheme is the possibility of extension with CHR features like guards and propagation, which will prove to be extremely useful.

An extension of Join-Patterns which introduces algebraic pattern matching in the matching of Join-Patterns is studied in [10]. Our approach generalizes this, as CHR matching semantics handles pattern matching over constraint (message) variables.

In the future, we intend to explore this relation further by implementing a prototype Join-Pattern system based on the CHR matching compilation scheme. Our CHR matching compilation scheme here is inherently single threaded, yet practical implementations would demand a system which is capable of executing matchings in parallel. Our works in a parallel implementation of CHR [15] would provide the framework for a prototype system based on parallel CHR matching. We also intend to investigate the implications of such implementations on performance, as well as on theoretical CHR properties (confluence, determinism).

Acknowledgments

We thank Simon Peyton Jones for suggesting to establish formal links between Join-Pattern and CHR matching compilation schemes. We thank Matthias Zenger for some initial discussions on how to encode Join-Calculus in Constraint Handling Rules. We thank the reviewers of CHR'08 for their useful comments on how to improve the paper.

References

1. Marco Alberti, Marco Gavanelli, Evelina Lamma, Federico Chesani, Paola Mello, and Paolo Torroni. Compliance verification of agent interaction: A logic-based software tool. *Applied Artificial Intelligence*, 20(2-4):133–157, April 2006.
2. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
3. Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM.
4. Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, page 22, Washington, DC, USA, 1999. IEEE Computer Society.
5. G. J. Duck. *Compilation of Constraint Handling Rules.* PhD thesis, The University of Melbourne, 2005.
6. F. Le Fessant and L. Maranget. Compiling join-patterns. In *HLCL '98: High-Level Concurrent Languages, volume 16(3) of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Sept. 1998.*, 1998.
7. Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.
8. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.
9. C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.
10. Qin Ma and Luc Maranget. Algebraic pattern matching in join calculus. *LMCS-4*, 1:7, 2008.
11. P.J.Stuckey and M. Sulzmann. A systematic approach in type system design based on constraint handling rules. Technical report, 2001.
12. T. Schrijvers. Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 435–436. Springer-Verlag, 2005.
13. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, 2004. ISSN 0939-5091.
14. M. Sulzmann and E. S. L. Lam. Haskell – Join – Rules. In Draft Proc. of IFL'07, September 2007.
15. M. Sulzmann and E. S. L. Lam. Parallel execution of multi set constraint rewrite rules. In proc. 10th International Symposium on Principles and Practice of Declarative Programming, July 2008.
16. Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *2nd Workshop on Constraint Handling Rules, Sitges, Spain*, pages 47–62, 2005.