TLP **18** (5–6): 759–805, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000078 First published online 23 May 2018

Parallelism, concurrency and distribution in constraint handling rules: A survey

THOM FRÜHWIRTH
Institute of Software Engineering and Programming Languages,
Ulm University
89069 Ulm, Germany
(*e-mail:* thom.fruehwirth@uni-ulm.de)

submitted 31 March 2017; revised 6 April 2018; accepted 10 April 2018

Abstract

Constraint Handling Rules (CHR) is both an effective concurrent declarative programming language and a versatile computational logic formalism. In CHR, guarded reactive rules rewrite a multi-set of constraints. Concurrency is inherent, since rules can be applied to the constraints in parallel. In this comprehensive survey, we give an overview of the concurrent, parallel as well as distributed CHR semantics, standard and more exotic, that have been proposed over the years at various levels of refinement. These semantics range from the abstract to the concrete. They are related by formal soundness results. Their correctness is proven as a correspondence between parallel and sequential computations. On the more practical side, we present common concise example CHR programs that have been widely used in experiments and benchmarks. We review parallel and distributed CHR implementations in software as well as hardware. The experimental results obtained show a parallel speed-up for unmodified sequential CHR programs. The software implementations are available online for free download and we give the web links. Due to its high level of abstraction, the CHR formalism can also be used to implement and analyse models for concurrency. To this end, the Software Transaction Model, the Actor Model, Colored Petri Nets and the Join-Calculus have been faithfully encoded in CHR. Finally, we identify and discuss commonalities of the approaches surveyed and indicate what problems are left open for future research.

KEYWORDS: parallelism, concurrency, distribution, constraint handling rules, declarative programming, concurrent constraint programming, semantics, rewriting, concurrency models.

1 Introduction

Parallelism has become an eminent topic in computer science again with the widespread arrival of multi-core processors. With the proliferation of mobile devices and the promises of the internet-of-things, distribution is an another major topic, intertwined with the parallelism. Parallel and distributed programming is known to be difficult. Declarative programming languages promise to ease the pain. This survey shows how parallelism and distribution are addressed in the declarative language Constraint Handling Rules (CHR).

Basic notions. Before we start with our survey, we shortly clarify the essential concepts at stake and introduce CHR. The technical terms of concurrency, parallelism and distribution have an overlapping meaning, and the processes are another central notion in this context. Due to their generality, they are hard to define precisely:

Concurrency allows for logically more or less independent computations, be they sequential or parallel. This abstract concept thus supports the modular design of independent program components that can be composed together.

Parallelism allows for computations that happen simultaneously, at the same time, thus hopefully improving performance. On the downside, sequential programs usually have to be rewritten to be able to run in parallel. With the arrival of multi-core processors, it has become a dominant computation model. The processors may have access to a shared memory to exchange information.

Distribution allows for program components that are located on physically distributed decentralized networked processors. Each processor has its own local memory (distributed memory). Personal computers, the internet and mobile devices have enforced this computational paradigm. Distribution introduces modularity and potential parallelism, but also the need for communication between the components.

Processes are programs that are executed independently but can interact with each other. Processes can either execute local actions or communicate, coordinate and synchronize by passing (sending and receiving) messages. Depending on context and level of abstraction, processes are also called threads, workers, tasks, activities or even agents.

Concurrency and distribution are easier with the declarative programming languages, since they are compositional: Different computations can be composed into one without unintended interference. Moreover, declarative languages offer a wealth of program analysis and reasoning techniques.

CHR. CHR is both an effective concurrent declarative constraint-based programming language and a versatile computational logic formalism (Frühwirth, 2009; Sneyers *et al.*, 2010; Frühwirth and Raiser, 2011; Frühwirth, 2015, 2016). CHR has its roots in constraint logic programming and concurrent constraint programming, but also integrates ideas from multi-set transformation and rewriting systems. While conceptually simple, CHR is distinguished by a remarkable combination of desirable features:

- A semantic foundation in classical logic as well as in linear logic (Betz, 2014).
- An effective and efficient sequential and parallel execution model (Frühwirth and Raiser, 2011).
- A proof that every algorithm can be expressed with best known time and space complexity (Sneyers *et al.*, 2009).
- Up to a million rule applications per second due to CHRs novel rule execution strategy based on lazy matching without conflict resolution (Van Weert, 2010).
- Guaranteed properties like the anytime algorithm and online algorithm properties (Abdennadher *et al.*, 1999).
- Program analysis methods for deciding essential properties like confluence and program equivalence (Abdennadher and Frühwirth, 1999).

The given references are meant to serve as starting points into the respective themes. One could continue with their references but also the papers that reference them.

Information on CHR can be found online at `http://www.constraint-handling-rules.org`, including news, tutorials, papers, bibliography, online demos and free downloads of the language.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Minimum example. Assume we would like to compute the minimum of some numbers, given as multiset `min(`n_1`), min(`n_2`),...,` `min(`n_k`)`. We interpret the constraint (predicate) `min(`n_i`)` to mean that the number n_i is a candidate for the minimum value. We make use of the following CHR rule that filters the candidates.

```
min(N) \ min(M) <=> N=<M | true.
```

The rule consists of a left-hand side, on which a pair of constraints has to be matched, a guard check `N=<M` that has to be satisfied, and an empty right-hand side denoted by `true`. In effect, the rule takes two `min` candidates and removes the one with the larger value (constraints after the \ symbol are to be removed). Starting with a given initial state, CHR rules are applied exhaustively, resulting in a final state. Note that CHR is a committed-choice language, i.e., there is no backtracking in the rule applications. Here the rule keeps on going until only one, thus the smallest value, remains as single `min` constraint. Note that the `min` constraints behave both as operations (removing other constraints) and as data (being removed). This abstraction is characteristic of the notion of constraint.

A *state* is a multi-set of constraints. In a sequential computation, we apply one rule at a time to a given state. A possible computation sequence is (where we underline constraints involved in a rule application)

$$\begin{array}{c} \underline{\texttt{min(1)}},\underline{\texttt{min(0)}},\texttt{min(2)},\texttt{min(1)} \mapsto \\ \underline{\texttt{min(0)}},\underline{\texttt{min(2)}},\texttt{min(1)} \mapsto \\ \underline{\texttt{min(0)}},\underline{\texttt{min(1)}} \mapsto \\ \texttt{min(0)} \end{array}$$

The final state is called *answer*. The remaining constraint contains the minimum value, in this case zero.

By the way, CHR insists on multi-sets so one can directly model resources as constraints, for example,

```
buy : cup \ euro, euro <=> coffee.
```

This rule expresses that we get a coffee for two euros if we have a cup. As we will see, there are also some semantics and implementations of CHR that are set-based.

Concurrency and parallelism in CHR. One of the main features of CHR is its inherent concurrency. Intuitively, in a parallel execution of a CHR program, rules can be applied to separate parts of a state in parallel. As we will see, CHR rules can even be applied in parallel to overlapping parts of a state, in principle without the need to change the program. This is referred to as *logical parallelism* or *declarative concurrency*.

The rule of `min` can be applied in parallel to different parts of the state:

$$\begin{array}{cc} \underline{\texttt{min(1)}},\underline{\texttt{min(0)}}, & \underline{\texttt{min(2)}},\underline{\texttt{min(1)}} \mapsto \\ \underline{\texttt{min(0)}}, & \underline{\texttt{min(1)}} \mapsto \\ \multicolumn{2}{c}{\texttt{min(0)}} \end{array}$$

We arrive at the answer in less computation steps than with the sequential execution.

The rule can also be applied in parallel to overlapping parts of the state, provided the overlap is not removed by any rule. For example, let the overlap be the constraint `min(0)`.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Then the three pairs `min(0), min(1)`, `min(0), min(1)` and `min(0) and min(2)` can be matched to different rule instances. (Note that we always match the same `min(0)`, but that we have two copies of `min(1)`.) These rules can be applied at the same time, since the common (overlapping) constraint `min(0)` is not removed.

$$\underline{\underline{\underline{\texttt{min(0)}}}},\ \underline{\texttt{min(1)}},\underline{\texttt{min(2)}},\underline{\texttt{min(1)}} \mapsto \texttt{min(0)}$$

So this is another, even shorter way to arrive at the same answer.

In CHR, concurrently executing processes are CHR constraints that communicate via a shared built-in constraint store. The built-in constraints take the role of (partial) messages and variables take the role of communication channels.

Guaranteed properties of CHR. First of all, the essential *monotonicity property* of CHR means that adding constraints to a state cannot inhibit the applicability of a rule. (Rule matching and guards check for presence of certain constraints, never absence.) Among other things, this monotonicity enables decidable program analyses and helps declarative concurrency. Most, but not all semantics that we introduce enjoy the monotonicity property.

Now assume that while the program runs, we add another constraint. It will eventually participate in the computation in that a rule will be applied to it. The answer will be as if the newly added constraint had been there from the beginning but ignored for some time. This property of a CHR program is called *incrementality* or *online algorithm property* and directly follows from monotonicity.

Furthermore, in CHR, we can stop the computation at any time and observe the current state as intermediate answer. We can then continue by applying rules to this state without the need to recompute from scratch. If we stop again, we will observe a next intermediate answer that is closer to the final answer. This property of a CHR program is called the *anytime algorithm property*. Note that by this description, an anytime algorithm is also an *approximation algorithm*, since intermediate answers more and more approximate the final answer.

Desirable property of confluence. This property of a program guarantees that any computation starting from a given initial state results in the same answer no matter which of the applicable rules are applied. There is a decidable, sufficient and necessary syntactic condition to analyze confluence of terminating programs and to detect rule pairs that led to non-confluence when applied. Among other things, confluence implies that rules can be applied in parallel, with the same result as any sequential computation, without the need for any modification of the given program. If on the other hand a program is not confluent, it may have to be rewritten to ensure proper parallel execution. This rewriting is aided by the method of completion, which automatically adds rules to a program to make it confluent (but may not terminate). An introduction into all these properties can be found in Frühwirth (2009). In the next section, we will discuss desirable properties that characterize the correspondence between different semantics of CHR.

Overview of the survey and its structure. The richness of topics in this survey, from formal semantics to hardware implementation and more, poses a challenge for the structure of this text. We decided to go from abstract to concrete while making sure concepts are introduced in sections before they are referred to in later sections.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Sections 2–4: Abstract parallel CHR semantics, example programs, extension by transactions. In the next section, we define abstract syntax and abstract operational semantics for CHR. One sequential transition describes rule applications, another one parallel transitions, a trivial third one connects the two. The essential correctness properties of monotonicity, soundness and serializability are introduced. In Section 3, we present common classic CHR example programs based on well-known algorithms. Often one rule suffices. All but one of the programs can be run in parallel without change. In Section 4, we extend abstract parallel CHR with transactions (CHRt), a popular and essential concept in concurrency.

Sections 5 and 6: Refining the parallel semantics and its implementation. In Section 5, we refine our abstract semantics by differentiating between a goal and a constraint store. The goal holds active constraints to execute them as processes in the operation, the constraint store holds inactive constraints as data. This implies that we now have to account for the in-activation (suspension) and re-activation (wake-up) of user-defined constraints. In Section 6, we describe an implementation of the refined semantics in Haskell using software transactions and the result of benchmark experiments showing parallel speed-ups.

Sections 7 and 8: Excursion: Set-based massive parallelism and hardware implementations. Section 7 introduces a more exotic abstract semantics that is massively parallel. It is also set-based. This theoretical model in the extreme case allows to find primes in constant time and to solve SAT problems in linear time. This comes with a cost: soundness only holds under a certain condition. We then move on to more mundane fast hardware implementations of the parallel CHR semantics introduced in Section 8 and again present some experimental evidence. It is typically one order of magnitude faster than the fastest software implementations. The translation scheme of the hardware implementations also applies to procedural languages like C and Java.

Section 9: Distribution in CHR. In Section 9, we discuss two distributed semantics for CHR, where the constraint store and computations are decentralized by introducing the notion of locations. Distribution requires a syntactic restriction on CHRs rule heads to ensure shared variables as communication channels among locations. The first semantics is informal and set-based, the second one full-fledged. Both semantics allow for propagation rules. Both semantics have been implemented.

Section 10: Concurrency models in CHR. Last but not the least, in Section 10, we shortly show the high-level encoding common formal models of concurrency in CHR on four concrete models: the Software Transaction Model, the Actor Model, Colored Petri Nets (CPNs) and the Join-Calculus have been faithfully embedded in CHR to enable comparison and further investigation by the program analyses available in CHR. The embeddings have been proven correct. Some embeddings are available online.

Sections 11 and 12: Discussion and conclusions. We end the paper with a discussion, directions for future work and in Section 11 with conclusions.

Within the sections, we also try to follow a standard structuring where applicable: We define the parallel or distributed semantics at hand and discuss its correspondence to the standard sequential CHR semantics. This is usually done by proving the properties of soundness and serializability, which are notions of correctness. Another property of interest is monotonicity, which is also enjoyed by the standard CHR. For software and hardware

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

implementations, we give free download links and we summarize experimental results found in the literature. We illustrate the approaches to semantics and implementation with additional examples.

For a better reading experience, we use the editorial we throughout. Of course, it refers to different authors in different sections of this paper.

2 Parallel abstract operational semantics of CHR

We will present the sequential equivalence-based abstract CHR semantics and extend it with parallelism. We just need a sequential transition describes rule applications, another one parallel transitions, a trivial third one that connects the two. We also introduce the three properties that prove the correctness of a given semantics with regard to a more abstract or a sequential semantics: monotonicity, soundness and serializability. We assume basic familiarity with the first-order predicate logic and the state transition systems. Readers familiar with CHR can skip most of this section. We start with some preliminaries.

2.1 Semantics of CHR and their properties

Structural Operational Semantics is a common inductive approach to describe the behavior of programming languages, in particular, concurrent ones. In Structural Operational Semantics, a *state transition system* specifies the computations. Transitions rewrite states and take the form of inference rules. All semantics of CHR, sequential or parallel, employ this approach.

Semantics for sequential CHR. They exist in various formulations and at various levels of refinement, going from the abstract to the concrete (refined) (Frühwirth, 2009; Betz *et al.*, 2010):

- The *very abstract semantics* (Frühwirth, 2009) is close to modus ponens of predicate logic.
- The *abstract semantics* (Abdennadher *et al.*, 1999) is the classical basis for CHR program analysis and its properties.
- The more recent *state-equivalence-based abstract semantics* (Raiser *et al.*, 2009) will be the starting point of our survey. We will extend it with parallelism.
- The *refined semantics* (Duck *et al.*, 2004) describes more concretely the actual behavior of the CHR implementations. All more concrete parallel semantics of CHR are based on it.

In addition, several alternative operational semantics for sequential CHR have been proposed.

Soundness and serializability. The *correctness* of a more refined semantics is shown by its *soundness* with regard to a more abstract semantics. This means that for each computation in the refined semantics, there is a corresponding computation in the abstract semantics. The converse (completeness) typically does not hold, because refined semantics are more concrete and thus rule out certain computations. When we introduce a parallel semantics for CHR, it will be related by soundness to a more abstract semantics and/or the sequential part of the semantics.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Actually, the *interleaving semantics* approach to concurrency is defined by the fact that for each possible parallel computation, there exists a corresponding sequential computation with the same result. The sequential computation uses interleaving of the different parallel computations. This means that a parallel computation step can be simulated by a sequence of sequential computation steps. This correspondence property is called *serializability (sequential consistency)*. Most semantics we discuss are correct in this way.

2.2 Abstract syntax of CHR

Constraints are relations, distinguished predicates of the first-order predicate logic. We differentiate between two kinds of constraints: *built-in (pre-defined) constraints* and *user-defined (CHR) constraints* which are defined by the rules in a CHR program. Built-in constraints can be used as tests in the guard as well as for auxiliary computations in the body of a rule. In this survey, besides the trivial constraint `true`, we will have syntactical equality = between logical terms and equations between arithmetic expressions.

Definition 2.1
A *goal* is a conjunction of built-in and user-defined constraints. A *state* is also a goal. Conjunctions are understood as *multi-sets* of their conjuncts. We will use letters such as $A, B, C, D, E, \ldots$ for goals and S and T for states.

A *CHR program* is a finite set of rules. A *(generalized) simpagation rule* is of the form

$$r : H_1 \backslash H_2 \Leftrightarrow C | B$$

where r : is an optional *name* (a unique identifier) of a rule. In the rule *head* (left-hand side), H_1 and H_2 are conjunctions of user-defined constraints, the optional *guard* $C|$ is a conjunction of built-in constraints, and the *body* (right-hand side) B is a goal.

In the rule, H_1 are called the *kept constraints*, while H_2 are called the *removed constraints*. At least one of H_1 and H_2 must be non-empty. If H_1 is empty, the rule corresponds to a simplification rule, also written

$$s : H_2 \Leftrightarrow C | B.$$

If H_2 is empty, the rule corresponds to a propagation rule, also written

$$p : H_1 \Rightarrow C | B.$$

Interestingly, most parallel semantics do not allow for propagation rules, while distributed semantics do. This will be discussed in Section 11.

Ground CHR. Most implementations and some semantics assume that variables are substituted by ground (variable-free) terms at run-time. This requirement can be captured by a common syntactic fragment of CHR: In *Ground CHR*, every variable in a rule (also) occurs in the head of the rule. We also say that the rule is *range-restricted*. This condition can be relaxed by allowing for local variables in the body of rule, provided they first occur in built-in constraints that always bound them to ground values at run-time (e.g., arithmetic functions). So given a ground initial states, all states in a computation will stay ground. As we will see, this greatly simplifies refined semantics and implementations, since then it is not necessary to account for the suspension and wake-up of user-defined constraints during

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

computations. It is worth noting that Ground CHR without propagation rules is still Turing-complete: It can implement a Turing machine with just one rule as we will see in Section 3.2.

2.3 Sequential abstract operational semantics of CHR

The semantics follows Raiser *et al.* (2009) and Betz (2014). It relies on a structural equivalence between states that abstracts away from technical details in a transition.

State equivalence. The equivalence relation treats built-in constraints semantically and user-defined constraints syntactically. Basically, two states are equivalent if they are logically equivalent (imply each other) while taking into account that user-defined constraints form a multi-set, i.e., multiplicities matter. For a state S, the notation S_{bi} denotes the built-in constraints of S and S_{ud} denotes the user-defined constraints of S.

Definition 2.2 (*State equivalence*)
Two states $S_1 = (S_{1\mathrm{bi}} \wedge S_{1\mathrm{ud}})$ and $S_2 = (S_{2\mathrm{bi}} \wedge S_{2\mathrm{ud}})$ are *equivalent*, written $S_1 \equiv S_2$, if and only if

$$\models \forall(S_{1\mathrm{bi}} \rightarrow \exists\bar{y}((S_{1\mathrm{ud}} = S_{2\mathrm{ud}}) \wedge S_{2\mathrm{bi}})) \wedge \forall(S_{2\mathrm{bi}} \rightarrow \exists\bar{x}((S_{1\mathrm{ud}} = S_{2\mathrm{ud}}) \wedge S_{1\mathrm{bi}}))$$

with $\bar{x}$ those variables that only occur in S_1 and $\bar{y}$ those variables that only occur in S_2.

The CHR state equivalence is defined by two symmetric implications and moreover syntactically equates the conjunctions of user-defined constraints as multi-sets. For example,

$$X{=}{<}Y \wedge Y{=}{<}X \wedge c(X,Y) \;\equiv\; X{=}Y \wedge c(X,X) \not\equiv X{=}Y \wedge c(X,X) \wedge c(X,X).$$

Transition. Using this state equivalence, the abstract CHR semantics is defined by a single transition that is the workhorse of CHR program execution. It defines the application of a rule. Let the rule $(r : H_1 \backslash H_2 \Leftrightarrow C|B)$ be a variant of a rule from a given program $\mathscr{P}$. A variant (renaming) of an expression is obtained by uniformly replacing its variables by fresh variables.

$$(\textbf{Apply}) \quad \frac{S \equiv (H_1 \wedge H_2 \wedge C \wedge G) \quad (r : H_1 \backslash H_2 \Leftrightarrow C|B) \in \mathscr{P} \qquad (H_1 \wedge C \wedge B \wedge G) \equiv T}{S \mapsto_r T}$$

Upper-case letters stand for (possibly empty) conjunctions of constraints in this section. The goal G is called *context* of the rule application. It is left unchanged.

In a *transition (computation step)* $S \mapsto_r T$, S is called *source state* and T is called *target state*. We may drop the reference to the program $\mathscr{P}$ and rule r to simplify the presentation.

If the source state can be made equivalent to a state that contains the head constraints and the guard built-in constraints of a variant of a rule, then we delete the removed head constraints from the state and add the rule body constraints to it. Any state that is equivalent to this target state is in the transition relation.

A *computation (derivation)* of a goal S in a program P is a connected sequence $S_i \mapsto S_{i+1}$ beginning with the *initial state (query)* S_0 that is S and ending in a *final state (answer, result)* or the sequence is *non-terminating (diverging)*. The notation $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Note that the abstract semantics does not account for termination of propagation rules: If a state can fire a propagation rule once, it can do so again and again, *ad infinitum*. This is called trivial non-termination of propagation rules. Most parallel semantics rule out propagation rules. Propagation rules and their termination will be discussed for distributed CHR in Section 9, though.

For the minimum example, here is a possible **(Apply)** transition from a state $S = (\min(0) \wedge \min(2) \wedge \min(1))$ to a state $T = (\min(0) \wedge \min(1))$:

$$\frac{\begin{array}{c} S \equiv (\min(X) \wedge \min(Y) \wedge X \leqslant Y \wedge (X = 0 \wedge Y = 2 \wedge \ \min(1))) \\ (\min(X) \backslash \min(Y) \Leftrightarrow X \leqslant Y | \text{true}) \\ (\min(X) \wedge X \leqslant Y \wedge \text{true} \wedge (X = 0 \wedge Y = 2 \wedge \min(1))) \equiv T \end{array}}{S \mapsto T}$$

2.4 Extension to parallel abstract semantics

We extend the abstract semantics by parallelism. We interpret conjunction as parallel operator. As we have seen for the minimum example, CHR rules can also be applied simultaneously to overlapping parts of a state, as long as the *overlap* (shared, common part) is not removed by any rule. Following Frühwirth (2005a), CHR parallelism with overlaps is called *strong*. It can be defined as follows, see also Chapter 4 in Frühwirth (2009).

(Strong) Parallelism (with overlap). We denote parallel transitions by the relation $\mapsto\!\!\!\!\!\Rightarrow$. The transition **(Intro-Par)** says that any sequential transition is also a parallel transition. The transition **(Parallel)** combines two parallel transitions using conjunction into a single parallel transition where the overlap E is kept.

$$(\textbf{Intro}-\textbf{Par}) \quad \frac{A \mapsto C}{A \mapsto\!\!\!\!\!\Rightarrow C}$$

$$(\textbf{Parallel}) \quad \frac{A \wedge E \mapsto\!\!\!\!\!\Rightarrow C \wedge E \qquad B \wedge E \mapsto\!\!\!\!\!\Rightarrow D \wedge E}{A \wedge B \wedge E \mapsto\!\!\!\!\!\Rightarrow C \wedge D \wedge E}$$

Again, back to the minimum example:

$$(\text{Parallel}) \quad \frac{\min(1) \wedge \min(0) \mapsto\!\!\!\!\!\Rightarrow \text{true} \wedge \min(0) \qquad \min(2) \wedge \min(0) \mapsto\!\!\!\!\!\Rightarrow \text{true} \wedge \min(0)}{\min(1) \wedge \min(2) \wedge \min(0) \mapsto\!\!\!\!\!\Rightarrow \text{true} \wedge \text{true} \wedge \min(0)}$$

Here the overlap is the goal $\min(0)$.

2.5 Properties: Monotonicity, soundness and serializability

The *monotonicity property* of CHR states that adding constraints to a state cannot inhibit the applicability of a rule (Abdennadher *et al.*, 1999). It is easy to see from the context of the sequential **(Apply)** transition and from the overlap of the **(Parallel)** transition that a rule can be applied in any state that contains its head and guard.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Theorem 2.1 (*Monotonicity of CHR*)
If $A \mapsto B$, then $A \wedge G \mapsto B \wedge G$. If $A \Mapsto B$, then $A \wedge E \Mapsto B \wedge E$.

The *correctness* of the abstract parallel semantics can be established by proving the following theorem.

Theorem 2.2 (*Soundness and serializability*)
If $A \Mapsto B$, then there exists a sequential computation $A \mapsto^* B$.

The essential aspect of the truth is that the (Parallel) transition can be simulated sequentially: If $A \wedge E \mapsto B \wedge E$ and $C \wedge E \mapsto D \wedge E$, then $A \wedge C \wedge E \mapsto S \mapsto B \wedge D \wedge E$, where S is either $A \wedge D \wedge E$ or $B \wedge C \wedge E$, i.e., the two transitions commute.

3 Parallel CHR example programs

These exemplary CHR programs are mostly folklore in the CHR community, see e.g. Chapters 2 and 7 in Frühwirth (2009). These are concise and effective implementations of classical algorithms and problems starting with finding primes, sorting, including Turing machines and ending with preflow-push and union-find (UF). Often one type of constraint and one rule will suffice, and we will not need more then six rules. Due to the guaranteed properties of CHR, these programs are also incremental anytime online approximation algorithms. Typically, they run in parallel without any need for modifying the program. An exception is UF, which is known to be hard to parallelize. We do it with the help of confluence analysis.

These sequential programs are in the subset of Ground CHR without propagation rules and can therefore be understood in all parallel semantics and executed in all parallel implementations surveyed without modification. On the other hand, most example programs may require some modification for distributed semantics and their implementations. As we will see, the experimental results report parallel speed-ups.

3.1 Algorithms of Erastothenes, Euclid, von Neumann, Floyd and Warshall

Here we introduce some classical algorithms over numbers and graphs. They are implemented as simple multi-set transformations reminiscent of the Chemical Abstract Machine. Typically, they can be implemented with one kind of constraint and a single rule in CHR that can be applied in parallel to pairs of constraints. Our running example of minimum falls into this category. These programs are confluent when run as intended, with ground goals. Correctness of each implementation can be shown by contradiction: Given the specified initial goal, if the resulting answer were not of the desired form, the rule would still be applicable.

Prime numbers. The following rule is like the rule for minimum, but the guard is different, more strict. In effect, it filters out multiples of numbers, similar to the Sieve of Erastothenes.

```
sift : prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

If all natural numbers from 2 to n are given, only the prime numbers within this range remain, since non-prime numbers are multiples of other numbers greater equal to 2. Obviously, the rules can be applied to the pairs of prime number candidates in parallel. In a parallel step, we can try to remove each prime by associating it with another prime such that the sift rule is applicable. This gives a maximum, linear parallel speed-up without the need to modify the program. This was confirmed experimentally for both a software and a hardware implementation (Lam, 2018; Triossi *et al.*, 2012).

Greatest common divisor (GCD). The following rule computes the greatest common divisor of natural numbers written each as `gcd(N)`.

```
gcd(N) \ gcd(M) <=> 0<N,N=<M | gcd(M-N).
```

The rule replaces M by the smaller number $M - N$ as in Euclid's algorithm. The rule maintains the invariant that the numbers have the same greatest common divisor. Eventually, if $N = M$, a zero is produced. The remaining non-zero `gcd` constraint contains the value of the gcd. The rules can be applied to the pairs of gcd numbers in parallel. Note that to any pair of gcd constraints, the rule will always be applicable. A parallel speed-up was observed in a hardware implementation (Triossi *et al.*, 2012), and even a super-linear speed-up in a software implementation (Lam, 2018).

Merge sort. The initial goal state contains arcs of the form `a->V` for each value `V`, where `a` is a given smallest (dummy) value.

```
msort : A->B \ A->C <=> A<B, B<C | B->C.
```

The rule only updates the first argument of the `arc` constraint, never the second. The first argument is replaced by a larger value and the two resulting arcs form a small chain `A->B, B->C`. The rule maintains the invariant that `A=<B`. So eventually, in each arc, a number will be followed by its immediate successor, and thus the resulting chain of arcs is sorted.

For sorting with optimal run-time complexity, we prefer merging arc chains of the same length. To this end, we precede each chain with its length, written as special arc `N=>FirstNode`. We also have to add a rule to initiate merging of chains of the same length:

```
N=>A, N=>B <=> A<B | N+N=>A, A->B.
```

In the initial goal, we now introduce constraints of the form `1=>V` for each value `V`. The rules can be applied to the pairs of arcs in parallel similar to the previous examples.

Floyd–Warshall all-pair shortest paths. Our implementation finds the shortest distance between all connected pairs of nodes in the transitive closure of a directed graph whose edges are annotated with non-negative distances.

```
shorten : arc(I,K,D1), arc(K,J,D2) \ arc(I,J,D3) <=>
                              D3>D1+D2 | arc(I,J,D1+D2).
```

Clearly, we can shorten arc distances in parallel by considering triples of arc constraints that match the head of the rule. In each parallel step, we can try to remove each arc by associating it with a corresponding pair of arc constraints and by checking if the rule is applicable then.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

3.2 Classical models and classical algorithms with statefulness

These algorithms about abstract problems are characterized by their *statefulness*, i.e., their essence is a state change, an update. While other declarative languages may not have an efficient way to update, CHR has a proven one by constant-time updating (i.e., removing and adding) user-defined constraints (Sneyers *et al.*, 2009).

Turing machine. Turing machine is the classical model of computability used in theoretical computer science. One rule suffices to implement it efficiently in CHR.

```
st(QI,SI,SJ,D,QJ) \ state(I,QI), cell(I,SI) <=> state(I+D,QJ), cell(I,SJ).
```

The state transition steps of the Turing machine are given as constraints `st(QI,SI,SJ,D,QJ)`: In the current state, `QI` reading tape symbol `SI`, write symbol `SJ` and move in direction `D` to be in state `QJ`. The direction is either left or right, we move along the cells of a tape. We represent cells as an array, so positions are numbers and the direction is either $+1$ or -1. A Turing machine with one tape is inherently sequential, since we can only be in one state at a time. Still parallelism can be employed to find the matching state transition constraint.

The implementation of the Turing machine shows Turing-completeness of the Ground CHR fragment with constants only and without propagation rules, actually with a single rule (Sneyers, 2008).

Dijkstras dining philosophers. In this classical problem in concurrency, several philosophers sit at a round table. Between each of them, a fork is placed. A philosopher either thinks or eats. In order to eat, a philosopher needs two forks, the one from his left and the one from his right. After a while, an eating philosopher will start to think again, releasing the forks and thus making them available to his neighbors again.

```
think_eat : think(X), fork(X), fork(Y) <=> Y =:= (X+1) mod n | eat(X).
eat_think : eat(X) <=> Y =:= (X+1) mod n | think(X), fork(X), fork(Y).
```

In the implementation, we assume a given number `n` of philosophers (and forks). They are identified by a number from zero to `n-1`. The rules are inverses of each other, the constraints simply switch sides.

The problem is to design a concurrent algorithm that is fair, i.e., that no philosopher will starve. Here we are mainly interested in the inherent parallelism of the problem. Disjoint pairs of neighboring forks can be used for eating in one parallel computation step. (For the experiments, time counters for eating and thinking were introduced into the program to introduce termination.)

Blocks world. Blocks world is a classical planning problem in Artificial Intelligence. It simulates robot arms re-arranging stacks of blocks.

```
grab  : grab(R,X),  empty(R), clear(X), on(X,Y) <=> hold(R,X), clear(Y).
putOn : putOn(R,Y), hold(R,X), clear(Y) <=> empty(R), clear(X), on(X,Y).
```

The *operation constraints* `grab` and `putOn` specify the action that is taken. The other constraints are *data constraints* holding information about the scenario. Operation constraints update the data constraints. The rule `grab` specifies that robot arm `R` grabs block `X` if `R` is empty and block `X` is clear on top and on block `Y`. As a result, robot arm `R` holds block `X` and block `Y` is clear. The rule `putOn` specifies the inverse action. The data constraints in the rule switch sides. At any time, only one of the actions is thus possible for a given

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

robot arm. Parallelism is induced by introducing several robot arms and multiple actions for them. Different robot arms can grab different clear blocks in parallel or put different blocks on different clear blocks in parallel.

3.3 Parallel preflow-push algorithm

Next we present two non-trivial algorithms, preflow-push and UF. Both algorithms are acknowledged in the literature to be hard to parallelize. To maintain the focus of the survey, we cannot explain these algorithms in detail.

The preflow-push algorithm (Goldberg and Tarjan, 1988) solves the maximum-flow problem. Intuitively, the problem can be understood as a system of connected water-pipes, where each pipe has a restricted given capacity. The system is closed except for one source and one sink valve. The problem now is to find the maximum capacity the system can handle from source to sink and to find the routes the water actually takes.

A flow network is a directed graph, where each edge is assigned a non-negative capacity. We want to find a maximum flow through the network from a source to a sink node under the capacity restrictions. The preflow-push algorithm moves flow locally between neighboring nodes until a maximum flow is reached.

In Meister (2007), we present and analyze a concise declarative parallel implementation of the preflow-push algorithm by just four rules. In the code listing below, comment lines start with the symbol %.

```
 % increase node height by one, remove minimum
lift : n(U,N), e(U,E) \ h(U,_), m(U,M,C)
       <=> U \= source, U \= sink, 0 < E, C =:= N+E | h(U,M+1).
 % replace K by HU in unchecked egde, insert minimum
up   : h(U,HU), h(V,HV) \ r(U,V,K)
       <=> HU =< HV, K < HU | m(U,HV,1), r(U,V,HU).
 % push flow downwards by one unit, insert minimum, reverse edge
push : h(U,HU), h(V,HV) \ e(U,EU), e(V,EV), r(U,V,_)
       <=>  0 < EU, HV < HU | e(U,EU-1), e(V,EV+1), m(V,HU,1), r(V,U,HV).
 % compute minimum for node, count for completeness
min  : m(U,M1,C1), m(U,M2,C2) <=> m(U,min(M1,M2),C1+C2).
```

The variable `U` stands for a node, `N` is its number of outward capacity edges, `E` is its current excess flow and `HU` is its current height. The constraint `m(U,M,C)` encodes a minimum candidate with value `M` for node `U`, where the counter `C` allows to detect if the minimum of all outward edges has been computed. The constraint `r(U,V,K)` encodes a residual edge from nodes `U` to `V` with remaining capacity `K`.

The implementation described in Meister (2007) simulates parallel computations sequentially using an interleaving semantics approach and time stamps for user-defined constraints. The active elements (nodes with excess flow) can be processed in parallel as long as their neighborhoods (set of nodes connected to them through an edge) do not overlap. In the simulation, we greedily, randomly and exhaustively apply as many rules as possible at a given time point t before progressing to time $t+1$. A speed-up in experiments with random graphs was consistently observed. The speed-up depends on the total amount of flow units, its distribution on disjoint nodes and the density of the flow network. A parallel speed-up was also confirmed in the experiments of Triossi *et al.* (2012).

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

3.4 Parallel union-find algorithm

This classical UF (also: disjoint set union) algorithm (Tarjan and Leeuwen, 1984) efficiently maintains disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. UF is acknowledged in the literature to be hard to parallelize.

In Frühwirth (2005a), we implement the UF algorithm in CHR with optimal time and space complexity and with the anytime online algorithm properties. This effectiveness is believed impossible in other pure declarative programming languages due to their inability to express destructive assignment in constant time. When the UF algorithm is extended by the rules that deal with the function terms (rational trees), it can be used for optimal complexity *unification* (Meister and Frühwirth, 2007). Last but not the least, a generalization of UF yields novel incremental algorithms for simple Boolean and linear equations (Frühwirth, 2006). See chapter 10 in Frühwirth (2009) for an overview of UF in CHR.

Parallelizing basic union-find. We only discuss the basic UF algorithm here, not the optimized one, since the former has been used in experiments (Sulzmann and Lam, 2008). In CHR, the *data constraints* `root` and arc `->` represent the tree data structure. With the UF algorithm come several *operation constraints*: `find` returns the root of the tree in which a node is contained, `union` joins the trees of two nodes and `link` performs the actual join.

```
union    : union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode : A->B  \ find(A,X) <=> find(B,X).
findRoot : root(A) \ find(A,X) <=> found(A,X).

linkEq   : link(X,Y), found(A,X), found(A,Y) <=> true.
linkRoot : link(X,Y), found(A,X), found(B,Y), root(A) \ root(B) <=> B->A.
```

The second argument of the find operation `find` holds a fresh variable as an identifier. When the root is found, it is recorded in the constraint `found`.

CHR confluence analysis[COMP: Please set the citation "Abdennadher and Frühwirth 2004; Abdennadher and Frühwirth 1998" as "Abdennadher and Frühwirth 1998, 2004" here.] (Abdennadher and Frühwirth, 1998; Abdennadher and Frühwirth, 2004) produces abstract states that reveal a deadlock: When we are about to apply the `linkRoot` rule, another link operation may remove one of the roots that we need for linking. From the non-confluent states, we can derive an additional rule for `found` that mimics the rule `findNode`: The `found` constraint now keeps track of the updates of the tree so that its result argument is always a root.

```
foundUpdate : A->B \ found(A,X) <=> found(B,X).
```

Linking for disjoint node pairs can now run in parallel. While this seems an obvious result, this semi-automatic confluence-based approach yields a non-trivial parallel variant of the optimized UF algorithm with path compression. Correctness of the parallelization is proven in both cases in Frühwirth (2005a). A parallel speed-up is reported in Lam (2018).

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

4 Parallel CHR with transactions

We now extend parallel CHR by transactions. Transactions will also be used for the implementation of parallel CHR in Section 6 and for encoding of a transaction-based concurrency model in CHR in Section 10.1.

Transactions. They alleviate the complexity of writing concurrent programs by offering entire computations to run atomically and in isolation. *Atomicity* means that a transaction either proceeds uninterrupted and successfully commits or has to rollback (undo its side-effects). In *optimistic concurrency control*, updates are logged and only committed at the end of a transaction when there are no update conflicts with other transactions. *Isolation* means that no intermediate update is observable by the another transaction. The highest level of isolation is *serializability*, the major correctness criterion for concurrent transactions: For each parallel execution, there is a sequential execution with the same result.

4.1 Transactions in parallel CHR

The paper (Schrijvers and Sulzmann, 2008) proposes CHRt as a conservative extension of CHR with atomic transactions. An atomic transaction is denoted as a meta-constraint `atomic(C)` where `C` is a conjunction of CHR constraints. Atomic transactions may appear in goals.

Example 4.1
Consider these CHR rules for updating a bank account:

```
balance(Acc,Bal), deposit(Acc,Amt)  <=> balance(Acc,Bal+Amt).
balance(Acc,Bal), withdraw(Acc,Amt) <=> Bal>Amt | balance(Acc,Bal-Amt).

transfer(Acc1,Acc2,Amt) <=> withdraw(Acc1,Amt), deposit(Acc2,Amt).
```

The `balance` constraint is a *data constraint*, and the `deposit` and `withdraw` constraints are *operation constraints*. The guard ensures that withdrawal is only possible if the amount in the account is sufficient. The transfer constraint rule combines deposit and withdrawal among two accounts.Now assume a transfer between two accounts:

```
balance(acc1,500), balance(acc2,0), transfer(acc1,acc2,1000)
```

We can execute the deposit, but we cannot execute the withdrawal due to insufficient funds. The transaction gets *stuck*. It has a deadlock and cannot proceed till the end. This is clearly not the desired behavior of a transfer. In CHRt, we can introduce a transaction to avoid this problem. The `transfer` constraint in the goal is wrapped by the meta-constraint `atomic`.

```
balance(acc1,500), balance(acc2,0), atomic(transfer(acc1,acc2,1000))
```

Now the incomplete transaction will be rolled back, no money will be transferred.

4.2 Abstract syntax and semantics of CHRt

We assume Ground CHR. We classify CHR constraints into *operation constraints* and *data constraints*. The distinction appeals to the intuitive understanding that operation constraints update data constraints. Thus, the head of a CHRt rule must contain exactly one

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

operation constraint. It requires one more transition for transactions. The **(Atomic)** transition executes any number of atomic transactions in parallel in a common context T of data constraints.

$$(\textbf{Atomic}) \qquad \frac{(T \wedge S_1 \wedge C_1 \mapsto^* T \wedge S_1'), \ldots (T \wedge S_n \wedge C_n \mapsto^* T \wedge S_n')}{T \wedge S_1 \wedge \ldots S_n \wedge atomic(C_1) \wedge \ldots atomic(C_n) \Rightarrow T \wedge S_1' \wedge \ldots S_n'}$$

In the transition, T, S_i and S_i' must be data constraints. The parallel step considers the separate evaluation of each C_i in isolation. The transactions only share the common data constraints T, which serves as a context. Note that each transaction may perform arbitrary many computation steps. Each transaction is fully executed until there are no operation constraints. It does not get stuck. So there are only data constraints in the target state.

4.3 Properties: Monotonicity, soundness and serializability

For CHRt programs, the following properties are proven to hold in Schrijvers and Sulzmann (2008).

Serializability. For each (atomic) transition with n concurrent transactions, there is a corresponding computation of n consecutive sequential (atomic) transitions each with only one transaction.

Soundness. For any computation in CHRt, there is a corresponding computation in CHR where the atomic wrappers are dropped.

Monotonicity. Although not proven in the paper, it follows from soundness and the context T of the (atomic) transition.

4.4 Encoding transactions in standard CHR

We want to execute CHRt in standard parallel CHR, i.e., without the (atomic) transition. The straightforward way is to execute atomic transactions only sequentially. Thus, we trivially guarantee the atomic and isolated execution of transactions. We identify two special cases where we can erase the atomic wrappers and still allow for parallel execution: bounded and for confluent transactions.

Bounded transactions. A *bounded transaction* is one that performs a finite, statically known number of transitions.`atomic(G) <=> G`. Then we unfold the rule (Frühwirth and Holzbaur, 2003; Frühwirth, 2005b; Gabbrielli *et al.*, 2013) until no more operation constraints appear in its body. Since the transaction is bounded, unfolding will eventually stop.

In the running example, we can replace the atomic transfer rule (since it is bounded) by the following rule:

```
balance(Acc1,Amt1),balance(Acc2,Amt2),atomic(transfer(Acc1,Acc2,Amt)) <=>
            Amt1>Amt | balance(Acc1,Amt1-Amt), balance(Acc2,Amt2+Amt).
```

The rule head expresses the fact that an atomic transfer requires exclusive access to both the accounts involved.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Built-In Constraint	e		
CHR Constraint	$c ::= p(\bar{t})$	Identified (CHR) Constraint	$nc ::= c\#i$
Goal Constraint	$g ::= c \mid e \mid nc$	Goal (Store)	$G ::= \uplus g$
Store Constraint	$sc ::= e \mid nc$	(Constraint) Store	$Sn ::= \bigcup sc$
State	$\sigma ::= \langle G, Sn \rangle$	Matched Constraints	$\delta ::= Sn \setminus Sn$

Fig. 1. Refined parallel CHR syntax.

Confluent transactions. The paper proves that if a CHRt program is confluent when we ignore atomic wrappers, then it can be executed in standard parallel CHR provided the initial goal never gets stuck (deadlocks). Confluence then guarantees that isolation is not violated.

Consider the example of the stuck transaction that attempts to overdraw an account. The withdraw rule can be fixed if we drop its guard (and hence allow negative balances):

```
balance(Acc,Bal), withdraw(Acc,Amt) <=> balance(Acc,Bal-Amt).
```

Any two consecutive transfers commute now. Regardless of the order they are performed in, they yield the same final result (even if the intermediate results differ). Hence, we can safely erase the atomic wrappers.

5 Refined parallel CHR semantics

A *refined* semantics for parallel CHR is developed and implemented in Sulzmann and Lam (2008), Lam and Sulzmann (2009), and Lam (2018). This semantics can be seen as a refinement of the parallel abstract semantics given before. In states, we now differentiate between the goal that holds active constraints to be processed, and the constraint store that holds inactive suspended constraints as data. This means that we have to account for the in-activation (suspension) and re-activation (wake-up) of user-defined constraints due to built-in constraints on shared variables. As before, the semantics is given in two parts, the sequential transitions and the parallel transitions and the properties of monotonicity, soundness and serializability are shown.

5.1 Syntax for refined parallel CHR

Figure 1 describes the syntax for the refined semantics. The notation $\bar{a}$ denotes a sequence of a's. We only consider built-in constraints that are syntactic equalities or arithmetic equations. To distinguish multiple occurrences (copies and duplicates) of CHR constraints, they are extended by a unique identifier. We call $c\#i$ an *identified constraint*. Conjunctions are modeled as (multi-)sets. Unlike in the abstract semantics, a state is now a pair: We distinguish between a goal (store) (a multi-set of constraints) and the (constraint) store (a set of built-in and identified CHR constraints). Correspondingly, there are goal and store constraints. We also introduce matched constraints that are pairs of store constraints which we will need as an annotation to transitions.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

$$\textbf{(Solve+Wake)}\quad \frac{W = WakeUp(e,Sn)}{\langle \{e\} \uplus G \mid Sn\rangle \overset{W\setminus\{\}}{\longmapsto} \langle W \uplus G \mid \{e\} \cup Sn\rangle}$$

$$\textbf{(Activate)}\quad \frac{i \text{ is a fresh identifier}}{\langle \{c\} \uplus G \mid Sn\rangle \overset{\{\}\setminus\{\}}{\longmapsto} \langle \{c\#i\} \uplus G \mid \{c\#i\} \cup Sn\rangle}$$

$$\textbf{(Apply-Remove)}\quad \frac{\begin{array}{c}\text{Variant of } (r \;:\; H'_P \backslash H'_S <=> t \mid B') \in \mathscr{P} \text{ such that}\\ \exists \phi \quad Eqs(Sn) \models \phi(t) \quad \phi(H'_P) = DropIds(H_P)\\ \phi(H'_S) = \{c\} \uplus DropIds(H_S) \quad \delta = H_P \backslash \{c\#i\} \cup H_S\end{array}}{\begin{array}{rl} & \langle \{c\#i\} \uplus G \mid \{c\#i\} \cup H_P \cup H_S \cup Sn\rangle\\ \overset{\delta}{\longmapsto} & \langle \phi(B') \uplus G \mid H_P \cup Sn\rangle\end{array}}$$

$$\textbf{(Apply-Keep)}\quad \frac{\begin{array}{c}\text{Variant of } (r \;:\; H'_P \backslash H'_S <=> t \mid B') \in \mathscr{P} \text{ such that}\\ \exists \phi \quad Eqs(Sn) \models \phi(t) \quad \phi(H'_S) = DropIds(H_S)\\ \phi(H'_P) = \{c\} \uplus DropIds(H_P) \quad \delta = \{c\#i\} \cup H_P \backslash H_S\end{array}}{\begin{array}{rl} & \langle \{c\#i\} \uplus G \mid \{c\#i\} \cup H_P \cup H_S \cup Sn\rangle\\ \overset{\delta}{\longmapsto} & \langle \phi(B') \uplus \{c\#i\} \uplus G \mid \{c\#i\} \cup H_P \cup Sn\rangle\end{array}}$$

$$\textbf{(Suspend)}\quad \frac{\textbf{(Apply-Remove)} \text{ and } \textbf{(Apply-Keep)} \text{ do not apply to } c\#i \text{ in } Sn}{\langle \{c\#i\} \uplus G \mid Sn\rangle \overset{\{\}\setminus\{\}}{\longmapsto} \langle G \mid Sn\rangle}$$

where
$$\begin{array}{lcl} Eqs(S) & = & \{e \mid e \in Sn, e \text{ is a built-in costraint}\}\\ DropIds(Sn) & = & \{c \mid c\#i \in Sn\} \uplus \{e \mid e \in Sn\}\\ WakeUp(e,Sn) & = & \{c\#i \mid c\#i \in Sn \wedge \phi \text{ m.g.u. of } Eqs(Sn) \wedge\\ & & \theta \text{ m.g.u. of } Eqs(Sn \cup \{e\}) \wedge \phi(c) \neq \theta(c)\}\end{array}$$

Fig. 2. Parallel CHR semantics (sequential part $\overset{\delta}{\longmapsto}$).

5.2 Sequential refined CHR semantics

The sequential part of the semantics in Figure 2 is a generalization of the refined CHR semantics of Duck *et al.* (2004). The semantics assumes generalized simpagation rules that are not propagation rules.

Constraints from the goal are executed one by one. A constraint currently under execution is called *active constraint*. It tries to apply rules to itself. To try a rule, the active constraint is matched against a head constraint of the rule. The remaining head constraints are matched with *partner constraints* from the constraint store. If there is such a complete matching and if the guard is satisfied under this matching, then the rule applies (fires). The constraints matching the removed constraints of the head are deleted atomically and the body of the rule is added to the state. Because of the role of the active constraint, we call the semantics *goal-based semantics*.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Transitions. A transition $\sigma \overset{\delta}{\mapsto} \sigma'$ maps the CHR state σ to σ' involving the CHR constraint goals in δ. The transition annotation δ holds the constraints that were matched with the rule head. It will be needed in the parallel part of the semantics.

The first transition **(Solve+Wake)** moves a built-in constraint, an equation or equality e, into the store and wakes up (re-activates) identified constraints in the store which could now participate in a rule application. This is the case when the built-in constraint effects variables in a user-defined constraint, because then the re-activated (woken) constraint may now be able to match a rule head and satisfy the guard of the rule. The function $WakeUp(e, Sn)$ computes a conservative approximation of the re-activated constraints, where m.g.u. denotes the most general unifier induced by a set of syntactic equations.

In transition **(Activate)**, a CHR constraint goal becomes active by annotating it with a fresh unique identifier and adding it to the store.

Rules are applied in transitions **(Apply-Remove)** and **(Apply-Keep)**. They are analogous, but distinguish if the active constraint $c\#i$ is kept or removed. In both cases, we seek for the missing partner constraints in the store, producing a *matching substitution* ϕ in case of success. The guard t must be logically entailed by the built-in constraints in the store under the substitution ϕ. Then we apply the rule instance of r by atomically removing the matching constraints H_S and adding the rule body instance $\phi(B)$ to the goal. We also record the matched identified constraints H_S and H_P in the transition annotation. In transition **(Apply-Remove)**, the matching constraints H_S include $c\#i$. Since $c\#i$ is removed, we drop it from both the goal and the store. In transition **(Apply-Keep)**, $c\#i$ remains and so can possibly fire further rules.

Finally, in transition **(Suspend)**, we put an active constraint to sleep. We remove the active identified constraint from the goal if no (more) rules apply to the constraint. Note that the constraint is kept suspended in the store and may be woken later on.

5.3 Extension to parallel refined CHR semantics

Figure 3 presents the parallel part of the refined operational semantics. It is a refinement of the parallel transition for the abstract semantics. We allow for multiple goal stores to be combined while the constraint store is shared among the parallel computations.

In the **(Intro-Par)** transition, we turn a sequential computation into a parallel computation. Transition **(Parallel-Goal)** parallelizes two parallel computations operating on the same shared store, if their matched constraints δ_1 and δ_2 do not have an overlap that involves removed constraints. They may overlap in the kept constraints. This makes sure that parallel computations remove distinct constraints in the store. The identifiers of constraints make sure that we can remove multiple but different copies of the same constraint. The matched constraints δ_1 and δ_2 are composed by the union of the kept and removed components, respectively, forming δ. Note that a context G is added to the goals in the resulting parallel transition, implying monotonicity.

5.4 Properties: Monotonicity, soundness and serializability

The following results are proven in the appendix of Lam and Sulzmann (2009).

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

$$\textbf{(Intro-Par)}\qquad \frac{\langle G \mid Sn\rangle \overset{\delta}{\longmapsto} \langle G' \mid Sn'\rangle}{\langle G \mid Sn\rangle \overset{\delta}{\longmapsto}_{||} \langle G' \mid Sn'\rangle}$$

$$\textbf{(Parallel-Goal)}\qquad \frac{\begin{array}{c}\langle G_1 \mid H_{S1} \uplus H_{S2} \uplus Sn\rangle \overset{\delta_1}{\longmapsto}_{||} \langle G_1' \mid H_{S2} \uplus Sn\rangle \\ \langle G_2 \mid H_{S1} \uplus H_{S2} \uplus Sn\rangle \overset{\delta_2}{\longmapsto}_{||} \langle G_2' \mid H_{S1} \uplus Sn\rangle \\ \delta_1 = H_{P1} \backslash H_{S1} \quad \delta_2 = H_{P2} \backslash H_{S2} \\ H_{P1} \subseteq Sn \quad H_{P2} \subseteq Sn \quad \delta = H_{P1} \uplus H_{P2} \backslash H_{S1} \uplus H_{S2} \\ H_{S1} \cap (H_{P2} \uplus H_{S2}) = \{\} \quad H_{S2} \cap (H_{P1} \uplus H_{S1}) = \{\}\end{array}}{\langle G_1 \uplus G_2 \uplus G \mid H_{S1} \uplus H_{S2} \uplus Sn\rangle \overset{\delta}{\longmapsto}_{||} \langle G_1' \uplus G_2' \uplus G \mid Sn\rangle}$$

Fig. 3. Parallel CHR semantics (parallel part $\overset{\delta}{\rightarrowtail}_{||}$).

Monotonicity holds for the goal store, but not for the constraint store. In an enlarged constraint store, the **(Suspend)** transition may not be possible anymore, because a new rule becomes applicable to the active constraint. The monotonicity is still sufficient though, because in the semantics, the constraint store is only populated via the goal store. *Serializability* holds: Any parallel computation can be simulated by a sequence of sequential computations in the refined semantics.

Furthermore, *soundness* holds: Any parallel computation has a correspondence in a suitable variant of the sequential abstract semantics. For the upcoming theorem, let us note that an *initial state* is of the form $\langle G, \{\}\rangle$, a *final state* is of the form $\langle \{\}, Sn\rangle$. Given a computation $\langle G \mid \{\}\rangle \rightarrowtail_{||}^{*} \langle G' \mid Sn\rangle$, the state $\langle G' \mid Sn\rangle$ is called a *reachable state*.

Theorem 5.1 (*Soundness*)
For any reachable state $\langle G \mid Sn\rangle$,

$$\begin{array}{ll}\text{if} & \langle G \mid Sn\rangle \rightarrowtail_{||}^{*} \langle G' \mid Sn'\rangle \\ \text{then} & (NoIds(G) \uplus DropIds(Sn)) \mapsto^{*} (NoIds(G') \uplus DropIds(Sn'))\end{array}$$

where $\mapsto^{*}$ denotes transitions in the sequential abstract semantics and where $NoIds = \{c \mid c \in G, c \text{ is a CHR constraint}\} \uplus \{e \mid e \in G, e \text{ is a built-in constraint}\}$.

6 Parallel CHR implementation in Haskell

The parallel refined semantics from the previous Section 5 has been implemented in the lazy functional programming language Haskell (Sulzmann and Lam, 2007, 2008; Lam and Sulzmann, 2007, 2009; Lam, 2018). Concretely, we use the Glasgow Haskell Compiler for implementing parallel Ground CHR because of its good support for shared memory and multi-core architectures. The implementation is available online for free download at `https://code.google.com/archive/p/parallel-chr/`. In principle, the system can be re-implemented in mainstream procedural languages such as C and Java. In this section, we give an overview of the implementation principles and the best experimental results,

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

details and more experiments with different settings can be found in the literature cited above.

6.1 Implementation principles

Our implementation follows the principles of standard sequential implementations of CHR where possible (Holzbaur *et al.*, 2005; Van Weert, 2010). The goal store is realized as a stack, the constraint store as a hash table. We implement common CHR optimizations, such as constraint indexing (hashing) and optimal join ordering for finding partner constraints with early guard scheduling.

Parallel goal execution must not remove constraints in overlaps that participate in several rule head matchings. We discuss two approaches of concurrency control to implement this kind of parallel rule-head matching, locking and transactions, before we settle for a hybrid approach.

Fine-grained lock-based parallel matching. Pessimistic concurrency control uses locking as the basic serialization mechanism. We restrict the access to each constraint in the shared store with a lock. When an active constraints finds an applicable rule, it will first try to lock its matching removed partner constraints. Kept constraints can be used by several rules simultaneously, so they need not be locked. Locking fails if any constraint in the complete rule head matching is already locked by another active constraint. If locking fails, the active constraint releases all its locks and tries to redo the rule application. If locking succeeds, the rule is applied. No unlocking is necessary since locked constraints are removed. This locking mechanism can avoid deadlocks and cyclic behavior using standard techniques for these problems such as timestamps or priorities.

Software transactional memory (STM). Optimistic concurrency control is based on transactions that can either commit or rollback and restart. We use the STM transactions provided in Haskell. The principles of transaction have been introduced in Section 4. The idea of STM is that atomic program regions are executed optimistically. That is, any read/write operations performed by the region are recorded locally and will only be made visible when the transaction is completed. Before making the changes visible, the underlying STM protocol will check for read/write conflicts with other atomically executed regions. If there are update conflicts among transactions, the STM protocol will randomly commit one of the atomic transactions and rollback the others (Shavit and Touitou, 1997). Committing means that the program updates become globally visible. Rollback means that we restart the program. The disadvantage of STM is that unnecessary rollbacks can happen. We will meet STM again in Section 10.1, when it is specified in CHR.

Hybrid STM-based locking scheme. In the implementation, we use both STM and traditional shared memory access locking techniques. The search for matching partner constraints is performed outside STM to avoid unnecessary rollbacks. When a complete rule head matching is found, we perform an STM procedure that we call *atomic rule-head verification*. It checks that all the constraints are still available and marks the constraints to be removed as deleted. These deleted constraints will be physically delinked from the constraint store, either immediately or later. Both behaviors can be implemented with standard concurrency primitives (such as compare-and-swap and locks).

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Number of Threads	1	2	4	8	Unbounded
Merge Sort	121%	94%	70%	52%	>200%
Gcd	109%	37%	18%	12%	123%
Parallel Union-Find	125%	82%	52%	32%	>200%
Blocks World	123%	77%	54%	39%	>200%
Dining Philosophers	119%	74%	49%	41%	>200%
Prime	115%	73%	46%	30%	155%
Fibonacci	125%	85%	59%	39%	>200%
Turing Machine	111%	63%	78%	70%	>200%

Fig. 4. Experimental results, with optimal configuration (on eight threaded Intel processor).

Thread pool. The naive way to implement a parallel CHR system is to spawn an active thread for each goal constraint in a state. Each thread tries to find its partner constraints. However, the thread and its later partner constraints would then compete for the same rule application. Moreover, the number of threads would be unbounded, as the number of constraints in a state is unbounded. Our implementation uses a bounded number of active threads. A *thread pool* maintains threads waiting for tasks to be allocated for parallel execution.

6.2 Experimental results

Experiments were performed on an Intel Core quad core processor with hyper-threading technology (that effectively allows it to run eight parallel threads). We measure the relative performance of executing with one, two, four, eight and an unbounded number of threads against our sequential CHR implementation in Haskell. The table in Figure 4 gives some exemplary results with these two optimizations: Each goal thread searches store constraints in a unique order to avoid matching conflicts and a special goal ordering for Merge Sort and Gcd is used (explained below).

There are several general observations to be made with regard to the number of threads. Executing with one goal thread is clearly inferior to the sequential implementation because of the wasted overhead of parallel execution. Executions with two, four and eight goal threads show a consistent parallel speed-up, with exception of the Turing machine. It is inherently single-threaded. Interestingly, we still obtain improvements from parallel execution of administrative procedures (for example, dropping of goals due to failed matching). Unbounded thread pooling is always slower than the sequential implementation. Furthermore, we observed a super-linear speed-up for the Gcd example with a *queue-based goal ordering* instead of the usual *stack-based ordering* in the goal store. In merge sort, we stack `->` constraints and queue just `=>` for optimal performance. Last but not the least, experiments also confirmed that there is a speed-up when a multi-core processor instead of a single-core processor is used.

7 Massively parallel set-based CHR semantics

A CHR semantics is *set-based* if conjunctions of constraints are considered as set instead of multi-set. In Raiser and Frühwirth (2010), we present a parallel execution strategy for

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

set-based CHR. The use of sets instead of multi-sets has a dramatic impact: It allows for multiple removals of constraints. This means that overlaps can be removed several times. We show that the resulting refined semantics is not sound in general anymore, but sound if the program is deletion-acyclic (i.e., when its simpagation rules do not allow for mutual removal of constraints). CHRmp programs for the computation of minimum, prime numbers and sorting can run in constant time, given enough processors. We describe a program for SAT solving in linear time.

7.1 Massively parallel set-based semantics CHRmp

As in the parallel abstract semantics, there are no restrictions on the syntax of CHR. Reconsider the essential (Parallel) transition of the abstract CHR semantics. Keep in mind that conjunctions of constraints are now interpreted as sets of constraints.

$$\text{(Parallel)}\quad \frac{A \wedge E \mapsto C \wedge E \qquad B \wedge E \mapsto D \wedge E}{A \wedge B \wedge E \mapsto C \wedge D \wedge E}$$

Consider the program

```
a <=> b,c.          a <=> b,d.
```

Then the following transition for the goal `a ∧ e` is possible in the set-based interpretation:

$$\frac{\mathtt{a} \wedge \mathtt{e} \mapsto \mathtt{b} \wedge \mathtt{c} \wedge \mathtt{e} \qquad \mathtt{a} \wedge \mathtt{e} \mapsto \mathtt{b} \wedge \mathtt{d} \wedge \mathtt{e}}{\mathtt{a} \wedge \mathtt{e} \mapsto \mathtt{b} \wedge \mathtt{c} \wedge \mathtt{d} \wedge \mathtt{e}}$$

This means that `a` is removed twice and `b` is only produced once.

When we generalize this observation, we see that overlaps between rule matchings can be removed arbitrary many times, leading to a kind of massive parallelism.

Refined CHRmp semantics. We refine this set-based semantics now. We assume CHR without propagation rules. In the body of a rule, we distinguish between CHR constraints B_c and built-in constraints B_b, and write B_c, B_b. A CHRmp state S (or T) is of the form $\langle \mathbb{G}; \mathbb{B} \rangle$, where the goal (store) $\mathbb{G}$ is a *set* (not multi-set) of constraints and the (built-in) constraint store $\mathbb{B}$ is a conjunction of built-in constraints. c and d are atomic constraints. We adapt the state equivalence $\equiv$ in the obvious way to CHRmp states.

Definition 7.1 (*Massively parallel transition*)
Given a CHRmp state $S = \langle \mathbb{G}; \mathbb{B} \rangle$. Let $\mathscr{R}$ be the smallest set such that for each rule variant $r : H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$, where $S \equiv \langle H_1 \cup H_2 \cup \mathbb{G}'; G \wedge \mathbb{B}' \rangle$ it holds that $(H_1, H_2, B_c, B_b, \mathbb{B}') \in \mathscr{R}$. We then define for any non-empty subset $R \subseteq \mathscr{R}$,

- the set of removed constraints $D = \{c \mid \exists(\_, H_2, \_, \_, \mathbb{B}') \in R, c \in \mathbb{G} : H_2 \wedge \mathbb{B}' \rightarrow c\}$;
- the set of added constraints $A = \{c \mid \exists(\_, \_, B_c, \_, \_) \in R : c \in B_c\}$;
- the conjunction of added built-in constraints $B = \bigwedge_{(\_,\_,\_,B_b,\mathbb{B}') \in R} \mathbb{B}' \wedge B_b$.

A *massively parallel transition (step)* of $S = \langle \mathbb{G}; \mathbb{B} \rangle$ using $\mathscr{R}$ is then defined as

$$\text{(Massive-apply)}\quad \langle \mathbb{G}; \mathbb{B} \rangle \twoheadrightarrow^R \langle (\mathbb{G} \setminus D) \cup A; \mathbb{B} \wedge B \rangle$$

If the specific set R is not of importance, we write $\twoheadrightarrow$ instead of $\twoheadrightarrow^R$.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

The idea is that in the set $\mathscr{R}$, we collect all possible rule applications and then we apply any subset of them at once in one parallel computation step. In this way, multiple removals of the same constraint are possible. In the extreme case, $R = \mathscr{R}$, so all possible rule applications are performed simultaneously. We call this *exhaustive parallelism*. With such an execution strategy, any CHRmp program is trivially *confluent*, because there are no conflicting rule applications. On the other hand, if R is a singleton set, only one rule is applied and we are back to sequential CHR.

Example 7.1
Reconsider the CHR program for computing prime numbers. Consider the state

$$S = \langle \{\text{prime}(2), \text{prime}(3), \text{prime}(4), \text{prime}(5), \text{prime}(X)\}; X{=}6 \rangle.$$

There are three possible rule applications, removing the non-prime numbers 4 and twice 6:

$$\mathscr{R} = \left\{ \begin{array}{l} (\{\text{prime}(N_1)\}, \{\text{prime}(M_1)\}, \emptyset, \top, X{=}6 \wedge N_1{=}2 \wedge M_1{=}4), \\ (\{\text{prime}(N_2)\}, \{\text{prime}(M_2)\}, \emptyset, \top, X{=}6 \wedge N_2{=}2 \wedge M_2{=}6), \\ (\{\text{prime}(N_3)\}, \{\text{prime}(M_3)\}, \emptyset, \top, X{=}6 \wedge N_3{=}3 \wedge M_3{=}6) \end{array} \right\}$$

We can now perform all three possible rule applications exhaustively parallel, i.e., $R = \mathscr{R}$, resulting in the following sets:

$$D = \{\text{prime}(4), \text{prime}(X)\}, \quad A = \emptyset,$$
$$B = (X{=}6 \wedge N_1{=}2 \wedge M_1{=}4) \wedge (X{=}6 \wedge N_2{=}2 \wedge M_2{=}6) \wedge (X{=}6 \wedge N_3{=}3 \wedge M_3{=}6)$$

This leads to the parallel transition:

$$S \twoheadrightarrow^{\mathscr{R}} \quad \langle \{\text{prime}(2), \text{prime}(3), \text{prime}(5)\}; X{=}6 \wedge B \rangle$$

Hence, a single parallel step is sufficient to find all prime numbers.

7.2 Example programs under exhaustive parallelism

We examine different algorithms written in CHR and the effect of executing these programs in CHRmp, in particular with exhaustive parallelism to achieve maximum speed-up.

Filter programs. Programs that only consist of rules whose body is `true` can be understood as filtering constraints. They can obviously be executed in constant time with exhaustive parallelism, given enough processors. The minimum and the prime program fall into this category. The `msort` rule of merge sort leads to a linear number of exhaustively parallel steps. It can be rewritten to achieve constant-time complexity. The experiments with the prime program using massive parallelism (see Section 8) (Triossi *et al.*, 2012) show a run-time improvement of about an order of magnitude over strong parallelism.

SAT solving. The SAT formula is given as a tree of its sub-expressions. The tree nodes are of the form `eq(Id,B)`, where `Id` is a node identifier and `B` is either a Boolean variable written `v(X)` or a Boolean operation (`neg, and, or`) applied to identifiers. Additionally, a `f(L,[])` constraint is required in the initial state, where `L` is a list of all n variables in the SAT formula.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

```
generate : f([X|Xs], A) <=> f(Xs,[true(X)|A]), f(Xs,[false(X)|A]).
assign   : f([],A) \ eq(T,v(X)) <=> true(X) in A | sat(T,A,true).
assign   : f([],A) \ eq(T,v(X)) <=> false(X) in A | sat(T,A,false).

sat(T1,A,S) \ eq(T,neg(T1)) <=> sat(T,A, neg S).
sat(T1,A,S1), sat(T2,A,S2) \ eq(T,and(T1,T2)) <=> sat(T,A, S1 and S2).
sat(T1,A,S1), sat(T2,A,S2) \ eq(T,or(T1,T2)) <=> sat(T,A, S1 or S2).
```

The `generate` rule generates, in n parallel steps, 2^n `f` constraints representing all possible truth assignments to variables as a list in its second argument. In the next parallel step (using the `assign` rules), all n Boolean variables in the given formula are assigned truth values for each assignment, represented by `sat` constraints.

The remaining three rules determine the truth values of all sub-expressions of the formula bottom-up. In each parallel step, the truth values of sub-expressions at a certain height of the tree are concurrently computed for all possible assignments of variables. Therefore, the number of parallel steps in this phase is bound by the depth of the formula.

A formula is in 3-DNF normal form if it is in disjunctive normal form (a disjunction of conjunctions of literals) and each clause contains at most three literals. Because of its bounded depth, a SAT problem given in 3-DNF normal form with n variables can be solved in linear time in n with this program under exhaustive parallelism, independent of the size of the formula.

7.3 Properties: Soundness under deletion-acyclicity

Soundness of CHRmp is not always possible as the following example shows.

Example 7.2
Consider the following rule that removes one of two differing constraints:

```
c(N) \ c(M) <=> N=\=M | true.
```

and the goal `c(1), c(2)`. There are two competing rule instances for application: one matches the two constraints in the given order and the other in reversed order. So if we apply both rules simultaneously under exhaustive parallelism, both constraints will be (incorrectly) removed.

In general, computations that allow for mutual removal of constraints are not sound in CHRmp. Soundness requires that the programs are deletion-acyclic, effectively ruling out mutual removal. A *deletion dependency pair* (c,d) means the kept constraint c is required to remove constraint d in a rule of the program. This is the case if c as an instance of a kept constraint and d is an instance of a removed constraint in the head of the rule.

Definition 7.2 (*Deletion dependency, deletion-acyclic*)
Given a CHRmp state $S = \langle \mathbb{G}; \mathbb{B} \rangle$. Then *deletion dependency* $\mathscr{D}(S)$ is a binary relation such that $(c,d) \in \mathscr{D}$ if and only if there exist $(H_1, H_2, B_c, B_b, \mathbb{B}') \in \mathscr{R}(S)$ and $c' \in H_1, d' \in H_2$ such that $c' \wedge \mathbb{B}' \rightarrow c$ and $d' \wedge \mathbb{B}' \rightarrow d$.

A CHRmp program $\mathscr{P}$ is *deletion-acyclic* if and only if for all S such that $S \twoheadrightarrow^{\mathscr{R}} T$ the transitive closure $\mathscr{D}(S)^+$ is irreflexive.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

In a deletion-acyclic program, we can simulate the CHRmp computation steps by a sequence of sequential rule applications in multi-set semantics, provided we initially have enough copies of the user-defined constraints and can remove them when needed. The latter is accomplished by so-called *set-rules* of the form

```
set-rule: c(X1,...Xn) \ c(X1,...Xn) <=> true.
```

for each CHR constraint `c/n` in the given program. These rules remove multiple occurrences of the same constraint.

The following soundness theorem requires a deletion-acyclic program and the set-rules (Raiser and Frühwirth, 2010). Let $\rightarrowtail$ be a sequential transition in a suitable variant of the usual multi-set CHR semantics.

Theorem 7.1 (*Soundness*)
Let $\mathscr{P}$ be a deletion-acyclic CHRmp program and $\mathscr{P}'$ be the CHR program $\mathscr{P}$ extended with set-rules. If $S = \langle \mathbb{G}; \mathbb{B} \rangle \twoheadrightarrow_{\mathscr{P}} T$, then there exists a multiset $\mathbb{G}'$ with $c \in \mathbb{G}' \Leftrightarrow c \in \mathbb{G}$ such that $S' = \langle \mathbb{G}'; \mathbb{B} \rangle \rightarrowtail^*_{\mathscr{P}'} T'$, where $c \in T' \Leftrightarrow c \in T$.

Example 7.3
Consider the initial goal `a` and the program

```
a <=> b,c.          a <=> b,d.          b,c,d <=> true.
```

Exhaustive parallelism leads to the set-based computation

$$\mathtt{a} \twoheadrightarrow \mathtt{b}, \mathtt{c}, \mathtt{d} \twoheadrightarrow \mathit{true}.$$

The sequential correspondence in the multi-set CHR program extended with set-rules is

$$\mathtt{a}, \mathtt{a} \rightarrowtail \mathtt{b}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rightarrowtail \mathtt{b}, \mathtt{c}, \mathtt{d} \rightarrowtail \mathit{true}.$$

The example can also be used to show that *serializability* in general does not hold for massively parallel set-based CHR. There is not sequential computation in CHRmp that can simulate the exhaustively parallel computation, since the first rule application will remove `a`, so either `b,c` or `b,d` can be produced sequentially, but not their union. Similarly, *monotonicity* does not hold.

8 Parallel hardware implementations of CHR

The work reported in Triossi *et al.* (2012) and Triossi (2011) investigates the compilation of CHR to the specialized hardware. The implementation follows the standard scheme for translating CHR into procedural languages. The compiler translates the CHR code into the low-level hardware description language VHDL, which in turn creates the necessary hardware using Field Programmable Gate Array (FPGA) technology. FPGA is a hardware consisting of programmable multiple arrays of logic gates. We also implement a hybrid CHR system consisting of a software component running a CHR system for sequential execution, coupled with hardware for parallel execution of dedicated rules in the program. The resulting hardware system is typically an order of magnitude faster than the fastest software implementation of CHR (in C).

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

8.1 Basic compilation of CHR to procedural languages

As preliminaries, we give the basic implementation scheme for Ground CHR in procedural languages like C and Java, but also VHDL. This translation scheme applies throughout this section. In Ground CHR, we do not need to wake-up constraints, because all variables are ground at run-time. A CHR rule can be translated into a procedure using the following simple scheme:

```
procedure(kept_head_constraints, removed_head_constraints) {
    if (head constraints not marked removed && head matching && guard check)
    then {remove removed_head_constraints; execute body constraints;}
}
```

The parameter list references the head constraints to be matched to the rule. In the procedure, we first check that the constraints have not been marked as removed. Then head matching is explicitly performed and then the guard is checked. If all successful, one removes the removed head constraints, executes the built-in constraints and then adds the body CHR constraints. Added constraints may overwrite removed head constraints for efficiency. Constraints that are removed and not overwritten are marked as deleted. Such a rule procedure is executed on every possible combination of constraints from the store, typically through a nested loop (that can be parallelized). This basic translation scheme corresponds to the abstract semantics, since it does not distinguish between active and suspended CHR constraints. It needs to be refined to be practical (Van Weert, 2010).

8.2 Compiling CHR to parallel hardware

Our compiler translates the CHR code into the low-level hardware description language VHDL, which in turn creates the necessary hardware using FPGAs. The architecture of FPGA hardware is basically divided into three parts: the internal computational units called configuration logic blocks, the Input/Output blocks that are responsible for the communication with all the other hardware resources outside the chip, and the programmable interconnections among the blocks called routing channels. In addition, there can be complex hardware blocks designed to perform higher level functions (such as adders and multipliers), or embedded memories, as well as logic blocks that implement decoders or mathematical functions.

CHR fragment with non-increasing rules. We assume Ground CHR. Since the hardware resources can only be allocated at compile time, we need to know the largest number of constraints that can occur in the constraint store during the computation. In *non-increasing rules*, the number of body CHR constraints added is not greater than the number of head constraints removed. Thus, the number of constraints in the initial goal provides an upper bound on the number of constraints during the computation. Hence, we only allow for non-increasing simpagation rules.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

CHR compilation hardware components. A program hardware block (PHB) is a collection of Rule Hardware Blocks (RHBs), each corresponding to a rule of the CHR program. A combinatorial switch assigns the constraints to the PHBs. In more detail:

Rule hardware block (RHB). In VHDL, the rule is translated into a single clocked process following the transformation scheme described above. Here, the parameters are input signals for each argument of the head constraints. Each signal is associated with a validity signal to indicate if the associated constraint has been removed. A concrete example is given below.

Program hardware block (PHB). The PHB makes sure that the RHBs keep applying themselves until the result remains unchanged for two consecutive clock cycles. Each rule is executed by one or more parallel processes that fire synchronously every clock cycle. The initial goal is directly placed in the constraint store from which several instances of the PHB concurrently retrieve the constraints.

Combinatorial switch (CS). The combinatorial switch sorts, partitions and assigns the constraints to the PHBs, ensuring that the entire constraint store gets exposed to the rule firing hardware. It acts as a synchronization barrier, allowing the faster PHBs to wait for the slower ones, then communicating the results between the blocks. It also re-assigns the input signals to make sure that all constraint combinations have been exposed to the rule head matching.

Strong parallelism with overlap. For a given kept constraint, multiple RHBs are used to try rules with all possible partner constraints. For the case of simpagation rules with one kept and one removed constraint, we introduce a hardware block that consists of a circular shift register which contains all the initial goal constraints. The first register cell contains the kept constraint and it is connected to the first input of all the RHBs, the rest of the register cells contain the potential partner constraints and are each connected to the second input of one RHB. Every time the PHBs terminate their execution, the new added constraints replace the removed ones. They shift registers until a non-removed constraint is encountered.

Example 8.1
Consider the rule for the greatest common divisor:

```
r :  gcd(N) \ gcd(M) <=> M>=N | gcd(M-N).
```

In Figure 5, we give an excerpt of the VHDL code produced for the above rule. There are two processes executed in parallel, one for each matching order, that correspond to two RHBs called `r_1` and `r_2`. The input parameters `gcd1` and `gcd2` are byte signals holding the numbers. `valid1s` and `valid2s` are bit signals. They are set to `0` if the associated constraint is removed. The shared variable `flag` is a bit. It is used to control the application of the two processes.

Massive parallelism. The set-based semantics CHRmp (see Section 7) allows multiple simultaneous removals of the same constraint. Our implementation eliminates the conflicts in the constraint removals by allowing different rule instances to work concurrently on distinct copies of the constraints. We provide all possible combinations of constraints to distinct parallel PHB instances in a single step. So the same constraint will be fed to several

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

```
r_1: process (..., gcd1s, gcd2s, valid1s, valid2s)
begin
  if ...      % checking and setting flags and parameters
    if (valid1s=1 and valid2s=1) then
      if gcd2s>=gcd1s then
        gcd2s <= gcd2s-gcd1s;
        flag := 1;
      else
        flag := 0;
      end if;
    end if;
  end if;
end process r_1;

r_2: process (..., gcd1s, gcd2s, valid1s, valid2s)
begin
  if ...      % checking and setting flags and parameters
    if (valid1s=1 and valid2s=1) then
      if flag=0 then
        if gcd1s>=gcd2s then
          gcd1s <= gcd1s-gcd2s;
        end if;
      end if;
    end if;
  end if;
end process r_2;
```

Fig. 5. Excerpt of VHDL code for GCD rule.

PHBs. Valid constraints are collected. A constraint is valid if no PHB has removed it. This is realized in hardware by AND gates. The improvement due to massive parallelism is about an order of magnitude for goals with a low number of constraints and it decreases with higher numbers of constraints. This is due to reaching the physical bounds of the hardware.

Experimental results. A few experiments were performed including the programs for minimum, prime numbers, GCD, merge sort, shortest-path and preflow-push (Triossi, 2011; Triossi *et al.*, 2012). Unfortunately, no tables with concrete performance numbers are given, just log-scale diagrams. From them, we can see the following. The FPGA implementations of CHR are at least one order of magnitude faster than the fastest software implementations of CHR. In the experiments, shortest-path and preflow-push showed a consistent parallel speed-up. Strong parallelism improves the performance, and massive parallelism improves it further by up to an order of magnitude for the prime example. In the examples, the code produced by the CHR-to-FPGA compiler is slower but within the same order of magnitude as handcrafted VHDL code.

Translation into C++ for CUDA GPU. Graphical Processing Units consist of hundreds of small cores to provide massive parallelism. Similar to the work on parallel CHR FPGA hardware, the preliminary work in Zaki *et al.* (2012) transforms *non-increasing Ground CHR rules* to C++ with CUDA in order to use a Graphical Processing Unit to fire the rules on all combinations of constraints. As a proof of concept, the scheme was encoded by

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

hand for some typical CHR examples. No experiments are reported. The constraint store is implemented as an array of fixed length consisting of the structures that represent CHR constraints. A CHR rule can be translated into a function in C++ using the basic procedural translation scheme. The rule is executed on every possible combination of constraints using nested for-loops. Finally, the code is rewritten for the CUDA library. The outer for-loop is parallelized for the thread pools of the Graphical Processing Unit.

9 Distribution in CHR

Before we introduce a full-fledged distributed refined semantics for CHR and its implementation, we set the stage by describing a distributed but sequential implementation of set-based CHR. This system is successfully employed in a verification system for concurrent software. Both semantics work with a syntactic subset of CHR where head constraints in rules must share variables in specific ways to enable locality of computations. Both semantics feature propagation rules, but they use different mechanisms to avoid their repeated re-application.

9.1 Distributed set-based goal stores in CHRd

CHRd (Sarna-Starosta and Ramakrishnan, 2007) is an implementation of a sequential *set-based refined* semantics for CHR with propagation rules. CHRd features a distributed constraint store.

Termination of propagation rules. There are basically two ways to avoid repeated application of propagation rules: Either they are not applied a second time to the same constraints or they do not add the same constraints a second time. Since we can remove constraints in CHR, usually the first option is chosen: We store the sequence of CHR constraint identifiers to which a propagation rule has been applied. It can be garbage-collected if one of the constraints is removed. This information is called a *propagation history*. CHRd replaces the check on the propagation history by an *occurrence check* on the constraint store. This can be justified by the set-based semantics.

Set-based refined semantics. Our set-based semantics closely follows the standard refined semantics (Duck *et al.*, 2004). The essential differences are as follows:

- The propagation history is dropped from the states.
- There is an additional transition to ensure a set-based semantics. It removes a constraint from the goal store before its activation, if it is already in the constraint store.
- There are additional transitions to avoid immediate re-application of a propagation rule. In the first transition, all head matching substitutions where the active constraint is kept are computed at once and all corresponding rule instances are added to the goal store. These rule instances are called *conditional activation events*.
- When a conditional activation event is processed, it is checked if the matching head constraints are still in the constraint store. If not, a second transition removes the event from the goal store. Otherwise, a third transition applies the rule instance by adding its body constraints to the goal store.

The semantics does not model the distribution of the CHRd constraint store.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Our set-based semantics is not always equivalent to the standard refined semantics. In the semantics, a propagation rule may fire again on a constraint that has been re-activated (woken). In the refined multi-set semantics, it will not be fired again. So a CHR program may not terminate with the set-based semantics, but with the refined semantics.

Distributed local constraint stores by variable indexing. Finding the partner constraints in head matching efficiently is crucial for the performance of a CHR system. If variables are shared among head constraints, we can use the corresponding arguments of the constraints for *indexing*. If the argument is an unbound variable at run-time, we store (a pointer to) the constraint as attribute of that variable. If the argument becomes bound (or even ground) at run-time, the constraint can be accessed from a hash table instead.

A conjunction of constraints is direct-indexed (connected) if all subsets of constraints share variables with the remaining constraints. In other words, it is not possible to split the constraints in two parts that do not share a variable.

Definition 9.1
The *matching graph* of a set C of constraints is a labeled undirected graph $G = (V,E)$, where $V = C$, and E is the smallest set such that $\forall c_1, c_2 \in V, \text{vars}(c_1) \cap \text{vars}(c_2) \neq \{\} \rightarrow (c_1, c_2) \in E$, where $\text{vars}(c)$ returns the set of variables in a constraint c. A rule R in a CHR program is said to be *direct-indexed (connected)* if the matching graph for its head constraints is connected. A CHR program is direct-indexed if all its rule heads are direct-indexed.

Clearly, head matching is significantly improved for direct-indexed programs. Instead of combinatorial search for matching partner constraints, constant-time lookups are possible with indexing. CHRd requires direct-indexed programs that only index on unbound variables. This permits the constraint store to be represented in a distributed fashion as a network of constraints on variables.

Any CHR program can be trivially translated to a direct-indexed program. We just have to add an argument to each CHR constraint that always contains the same shared variable. For example, the direct-index rule for minimum is

```
min(X,N) \ min(X,M) <=> N=<M | true.
```

With the help of the new variable, we can distinguish between different minima. In general, this technique can be used to localize computations.

Implementation and experimental results. We have an implementation of ground CHRd in the Datalog fragment of Prolog, where terms are constants only. Our implementation has been integrated into XSB, a Prolog programming system with tabling. It can be obtained online with a free download from `http://xsb.sourceforge.net`. CHRd performs significantly better on programs using tabling, and shows comparable results on non-tabled benchmarks. This indicates that constraint store occurrence checks can be done as efficiently as propagation history checks while avoiding the maintenance of a propagation history.

Verification of multi-threaded applications. The paper (Sarna-Starosta *et al.*, 2007) describes an approach for checking for deadlocks in multi-threaded applications based on the concurrency framework SynchroniZation Units MOdel (Szumo) (Sarna-Starosta, 2008). The framework associates each thread with a synchronization contract that

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

governs how it must synchronize with other threads. At run-time, schedules are derived by negotiating contracts among threads.

The Szumo system includes a constraint solver written in CHRd encoding the synchronization semantics of thread negotiation. The verification system performs a reachability analysis: It constructs execution paths incrementally until either a deadlock is detected or further extending the path would violate a synchronization contract.

With Szumo, we analyzed an implementation of the dining philosophers problem, where no deadlock was found. We verified the in-order message delivery property of an n-place FIFO buffer. We also analyzed Fischers protocol, a mutual-exclusion protocol that is often used to benchmark real-time verification tools. There we employed CHRd to specify a solver for the clock constraints.

9.2 Distributed parallel CHRe and its syntax

The paper (Lam and Cervesato, 2013) introduces a decentralized distributed execution model consisting of an ensemble of computing entities, each with its own local constraint store and each capable of communicating with its neighbors: In CHRe, rules are executed at one location and can access the constraint stores of its immediate neighbors. We have developed a prototype implementation of CHRe in Python with MPI (Message Passing Interface) as a proof of concept and demonstrated its scalability in distributed execution. It is available online for free download at `https://github.com/sllam/msre-py`.

Syntax of CHRe. We assume Ground CHR. CHRe introduces locations.

Definition 9.2
All user-defined constraints in a program must be explicitly localized. A *location* l is a term (typically an unbound variable or constant) that annotates a CHR constraint c, written as $[l]c$. A location l is *directly connected* to a location l' if there is a constraint $[l]c$ at location l such that $l \in \mathrm{vars}(c)$.

We are interested in rules that can read data from up to n of their immediate neighbors, but can write to arbitrary neighbors. We therefore define *n-neighbor restricted (star-shaped) rules* (which are a subclass of direct-indexed rules introduced in CHRd). The rule head refers to directly connected locations in a star topology. At the center of the star is the *primary location*.

Definition 9.3
A CHR rule with $n+1$ head constraints is *n-neighbor restricted (star-shaped)* if and only if there is a dedicated location called *primary location* and n —em neighbor locations in the rule head satisfying the following conditions:

- The primary location is directly connected to each of its n neighbor locations.
- If a variable is shared between constraints at different locations, it also must occur in the primary location.
- Each constraint in the guard shares variables with at most one neighbor location.

This definition ensures that computation can be structured and distributed by considering interactions between the primary location and each neighboring location separately.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Example 9.1
This variant of the Floyd–Warshall algorithm computes all-pair shortest paths of a directed graph in a distributed manner.

```
base  : [X]arc(Y,D) ==> [X]path(Y,D).
elim  : [X]path(Y,D1) \ [X]path(Y,D2) <=> D1<D2 | true.
trans : [X]arc(Y,D1), [Y]path(Z,D2) ==> X\=Z | [X]path(Z,D1+D2).
```

We distinguish between arcs and paths. `[X]path(Y,D)` denotes a path of length `D` from `X` to `Y`. The rules `base` and `elim` are 0-neighbor restricted (local) rules because their left-hand sides involve constraints from exactly one location. Rule `trans` is a 1-neighbor restricted rule since its left-hand side involves `X` and a neighbor `Y`. We see that `X` is the primary location of this rule because it refers to location `Y` in an argument.

9.3 Refined semantics of CHRe

Before we discuss the refined semantics, we shortly mention the abstract semantics of CHReto introduce the basic principles.

Abstract distributed CHRe semantics for n-neighbor restricted rules. Each location has its own goal store. Based on the standard abstract CHR semantics, we introduce abstract *ensemble* states, which are sets of local stores G_k where G is a goal and k a unique location name. In the adapted **(Apply)** transition, each of the locations in an n-neighbor rule provides a partial match in their stores. If the matchings can be combined and if the guard holds, we add the rule body goals to their respective stores. We show *soundness* with respect to the standard CHR abstract semantics, where locations are encoded as an additional argument to each CHR constraint.

Refined distributed CHRe semantics for 0-neighbor restricted rules. We extend the standard CHR refined semantics to support decentralized incremental multi-set matching for 0-neighbor restricted rules.

Localized states. In CHRe, an *ensemble* Ω is a set of localized states. A *localized state* is a tuple $\langle \vec{U}; \vec{G}; \bar{S}; \bar{H} \rangle_k$, where

- the *Buffer* $\vec{U}$ is a queue of CHR constraints that have been sent to a location,
- the *Goal Store (Execution Stack)* $\vec{G}$ is a stack of the constraints to be executed,
- the *Constraint Store* $\bar{S}$ is a set of identified constraints to be matched,
- the *Propagation History* $\bar{H}$ is a set of sequences of identifiers of constraints that matched the head constraints of a rule,
- the state is at *location* k.

To add a further level of refinement, an *active occurrenced CHR constraint* $c(\bar{x})\#i:j$ is an identified constraint that is only allowed to match with the jth occurrence of the constraint predicate symbol c in the head of a rule of a given CHR program $\mathscr{P}$.

To simplify the presentation of the semantics, we assume static locations: For all locations occurring in a computation, there is a localized state (possibly with empty components) in the ensemble.

Localized sequential transitions. Figure 6 shows the sequential transitions for a single location.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

(Flush)
$$\frac{\vec{U} \neq \{\}}{\Omega, \langle \vec{U}; \{\}; \bar{S}; \bar{H} \rangle_k \mapsto \Omega, \langle \{\}; \vec{U}; \bar{S}; \bar{H} \rangle_k}$$

(MoveLoc)
$$\frac{}{\Omega, \langle \vec{U}; ([k']c, \vec{G}); \bar{S}; \bar{H} \rangle_k, \langle \vec{U}; \vec{G}; \bar{S}; \bar{H} \rangle_{k'} \mapsto \Omega, \langle \vec{U}; \vec{G}; \bar{S}; \bar{H} \rangle_k, \langle ((\vec{U}, [c]); \vec{G}; \bar{S}; \bar{H} \rangle_{k'}}$$

(DropLoc)
$$\frac{}{\Omega, \langle \vec{U}; ([k]c, \vec{G}); \bar{S}; \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; (c, \vec{G}); \bar{S}; \bar{H} \rangle_k}$$

(Activate)
$$\frac{d \text{ is a fresh identifier}}{\Omega, \langle \vec{U}; (c, \vec{G}); \bar{S}; \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; (c\#d : 1, \vec{G}); (\bar{S}, c\#d); \bar{H} \rangle_k}$$

(Remove)
$$\frac{\begin{array}{c}\text{Variant of } (r \ : \ [l]H'_P \backslash [l]H'_S <=> C \mid B) \in \mathscr{P} \text{ such that } \models \phi(C) \quad k = \phi(l) \\ \phi(H'_P) = DropIds(H_P) \quad \phi(H'_S) = \{c\} \cup DropIds(H_S)\end{array}}{\Omega, \langle \vec{U}; (c\#d : i, \vec{G}); (\bar{S}, H_P, H_S, c\#d); \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; (\phi(B), \vec{G}); (\bar{S}, H_P); \bar{H} \rangle_k}$$

(Keep)
$$\frac{\begin{array}{c}\text{Variant of } (r \ : \ [l]H'_P \backslash [l]H'_S <=> C \mid B) \in \mathscr{P} \text{ such that } \models \phi(C) \quad k = \phi(l) \\ \phi(H'_S) = DropIds(H_S) \quad \phi(H'_P) = \{c\} \cup DropIds(H_P) \quad h = (r, Ids(H_P, H_S)), h \notin \bar{H}\end{array}}{\Omega, \langle \vec{U}; (c\#d : i, \vec{G}); (\bar{S}, H_P, H_S, c\#d); \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; (\phi(B), c\#d : i, \vec{G}); (\bar{S}, H_P, c\#d); (\bar{H}, h) \rangle_k}$$

(Suspend)
$$\frac{\text{(Remove) and (Keep) do not apply for } c\#d : i, \text{ occurrence } i \text{ exists}}{\Omega, \langle \vec{U}; (c\#d : i, \vec{G}); \bar{S}; \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; (c\#d : (i+1), \vec{G}); \bar{S}; \bar{H} \rangle_k}$$

(Drop)
$$\frac{i \text{ is not an occurrence in the program } \mathscr{P}}{\Omega, \langle \vec{U}; (c\#d : i, \vec{G}); \bar{S}; \bar{H} \rangle_k \mapsto \Omega, \langle \vec{U}; \vec{G}; \bar{S}; \bar{H} \rangle_k}$$

Fig. 6. The sequential part of the refined CHRe semantics for 0-neighbor restricted rules.

- The **(Flush)** transition step applies if the goal store is empty and the buffer is non-empty. It moves all buffer constraints into the goal store.

The transitions **(DropLoc)** and **(MoveLoc)** apply if the first constraint in the goal store of location k is one for location $[k']c$. They deliver constraint $[k']c$ to location k'.

- The **(MoveLoc)** transition applies if k' is distinct from k and there exists a location k'. It it strips the location $[k]$ away and sends constraint c to the buffer of k'.
- The **(DropLoc)** transition applies if k' is the same as k. The location $[k]$ is dropped.

The remaining transitions apply to a location as to a state in the standard refined semantics. Buffers are ignored and remain unchanged. The transitions model the activation of a constraint, the application of rules to it, and its suspension if no more rule is applicable.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

$$\textbf{(Intro-Par)} \qquad \frac{\Omega \mapsto \Omega'}{\Omega \Mapsto \Omega'}$$

$$\textbf{(Parallel-Ensemble)} \qquad \frac{(\Omega_1, \Omega_2) \Mapsto (\Omega_1', \Omega_2) \qquad (\Omega_1, \Omega_2) \Mapsto (\Omega_1, \Omega_2')}{(\Omega_1, \Omega_2) \Mapsto (\Omega_1', \Omega_2')}$$

Fig. 7. The parallel part of the refined CHRe semantics for 0-neighbor restricted rules.

These transitions are as in the standard refined semantics of CHR, except that here we take care of locations and handle a propagation history.

- In the **(Activate)** transition, a CHR constraint c becomes active (with first occurrence 1) and is also introduced as identified constraint into the constraint store.
- The **(Remove)** transition applies a rule where the active constraint is removed. There is a substitution θ under which constraints from the constraint store match the head of the rule and satisfy its guard (written $\models \theta \wedge G$).
  The auxiliary function DropIds removes the identifiers from identified constraints.
- The **(Keep)** transition is like the **(Remove)** transition except that the active constraint c matches a kept constraint and it is checked if the application of the resulting rule instance has not been recorded in the propagation history. If so, the active constraint is kept and remains active. The propagation history is therefore updated. (It remains unchanged in all other transitions.) The function Ids returns the identifiers of identified constraints.
- In the **(Suspend)** transition, the active constraint cannot be matched against its occurrence in the rule head. One proceeds to the next occurrence in the rules of the program. This makes sure that rules are tried in the order given in the program.
- The **(Drop)** transition, if there is no more occurrence to try, removes the active constraint the goal store, but it stays suspended in the constraint store.

Localized parallel transitions. Figure 7 shows the parallel transitions. They are particularly simple. As usual, the transition **(Intro-Par)** says that any sequential transition is a parallel transition. Transition **(Parallel-Ensemble)** allows to combine two independent transitions on non-overlapping parts of the state (ensembles, i.e., sets of disjoint locations) into one parallel transition. This means that computation steps on different localized states can be executed in parallel.

9.4 Properties: Monotonicity, soundness and serializability

In the refined CHRe semantics, monotonicity holds with respect to locations, this means computations can be repeated in any larger context of more locations. Serializability holds in that every parallel CHRe computation can be simulated using sequential CHRe transitions. We also prove soundness of the refined CHRe semantics with respect to the abstract CHRe semantics.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

We say that a CHRe program is *locally quiescent (terminating)* if given a reachable state, we cannot have any infinite computation sequences that do not include the (Flush) transition. Hence, local quiescence guarantees that each location will eventually process the constraints delivered to its buffer.

Serializability and soundness of the encoding holds for quiescent programs: computations between commit-free states of 0-neighbor restricted encodings have a mapping to computations of the original 1-neighbor restricted program.

The corresponding theorems and their detailed proofs can be found in the appendix of Lam and Cervesato (2013).

9.5 Encoding 1*- and* n*-neighbor rules in local rules*

We give an encoding of the more general 1-neighbor restricted rules into local, i.e., 0-neighbor restricted rules. We can do the same for n-neighbor restricted rules. In this way, a programmer can use n-neighbor rules while the translation generates the necessary communication and synchronization between locations. The encodings are a block-free variation of a two-phase commit consensus protocol between locations.

Two-phase-commit consensus protocol. The protocol consists of two phases:

- *Commit-request phase (voting phase).* The coordinator process informs all the participating processes about the transaction and to vote either commit or abort. The processes vote.
- *Commit phase.* If all processes voted commit, the coordinator performs its part of the transaction, otherwise aborts it. The coordinator notifies all processes. The processes then act or abort locally.

The standard protocol can block if a process waits for a reply. Not so in the variation we use.

Encoding 1-neighbor restricted programs. According to the following scheme, we translate each 1-neighbor restricted rule of the form

```
r : [X]Px, [X]Px', [Y]Py \ [X]Sx, [Y]Sy <=> Gx,Gy | Body.
```

In the head, `Px` are the persistent constraints and `Px'` are the non-persistent constraints. Constraints are *persistent* if they are not removed by any rule in the program. In the guard, `Gx` contains only variables from location `X`. In the rule scheme below, `XYs` contains all variables from the rule head, and `Xs` only the variables from location `x`.

```
 % Commit-Request Phase
 % match and send request to neighbor location
request : [X]Px,[X]Sx ==> Gx | [Y]r_req(Xs).
 % match and send commit to primary location
vote : [Y]Py,[Y]Sy \ [Y]r_req(Xs) <=> Gy | [X]r_vcom(XYs). % if Sx non-e.
vote : [Y]Py,[Y]Sy,  [Y]r_req(Xs) ==> Gy | [X]r_vcom(XYs). % if Sx empty

 % Commit Phase
 % remove non-persistent constraints at primary location and send commit
commit :  [X]Px \ [X]Px',[X]Sx, [X]r_vcom(XYs) <=> [Y]r_commit(XYs).
 % remove at neighbor location, add non-persistent and body constraints
```

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

```
act   : [Y]Py \ [Y]Sy, [Y]r_commit(XYs) <=> [X]Px', Body.
 % otherwise abort, re-introduce removed constraints at primary location
abort : [Y]r_commit(XYs) <=> [X]Px', [X]Sx.
```

The rule scheme uses different `vote` rules depending on the emptiness of `Sx`. If `Sx` is empty, it should be possible to remove several instances of `Sy` with the same request. Note that the rule scheme requires a refined semantics where rules are tried in the given order, because we have to make sure that rule `act` is tried before the abort rule `abort`.

The rule scheme implements an *asynchronous and optimistic consensus protocol* between two locations of the ensemble. It is asynchronous because neither primary nor neighbor location ever block or busy-wait for responses. Rather they communicate asynchronously via the protocol constraints, while potentially interleaving with other computations. The temporary removal of non-persistent constraints in the rule scheme ensures that the protocol cannot be interfered with. It is optimistic because non-protocol constraints are only removed after both locations have independently observed their part of the rule head instance. It is possible that some protocol constraints are left if the transaction did not commit, but these can be garbage-collected.

We can generalize the above encoding to n-neighbor restricted rules.

CoMingle. This new programming language can be characterized as an extension of CHRe for distributed logic programming (Lam *et al.*, 2015; Cervesato *et al.*, 2016). There is a prototype on the Android operating system for mobile devices, see `https://github.com/sllam/CoMingle`. One application was built both using CoMingle and by writing traditional code: The former was about one tenth of the size of the latter without a noticeable performance penalty.

10 Models of concurrency in CHR

Theoretical and practical models of concurrency have been encoded in CHR. Such an effective and declarative embedding holds many promises: It makes theoretical models executable. It can serve as executable specification of the practical models. One can toy with alternative design choices. The implementations can be formally verified and analyzed using standard and novel CHR analysis techniques. Last but not the least, it allows to compare different models on a common basis.

We will shortly introduce some common models of concurrency by their implementation in CHR: STM, Colored Petri Nets, actors and join-calculus. Typically, soundness and completeness results will prove the correctness of these embeddings.

10.1 Software transactional memory (STM)

We have already seen the description of STM and its use to implement parallel CHR in Haskell in Section 6. Now we do it the other way round. For the STM model, as a starting reference, see Shavit and Touitou (1997), for a high-level description, see Guerraoui and Kapalka (2008). The paper (Sulzmann and Chu, 2008) gives a rule-based specification of Haskell's STM in parallel CHR which naturally supports the concurrent execution of transactions.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

We classify CHR constraints once more into operation constraints and data constraints. We assume CHR rules where the head contains exactly one operation constraint and the body contains at most one operation constraint.

Shared memory operations. We first model shared memory and its associated read and write operations in CHR.

```
read  : cell(L,V1) \ read(L,V2) <=> V1=V2.
write : cell(L,V1), write(L,V2) <=> cell(L,V2).
```

`L` is a location identifier and `V1` and `V2` are values. `cell` is a data constraint, `read` and `write` are operation constraints. The `write` rule performs a destructive assignment to update the value of the cell. With indexing and in-place constraint updates, the compiled rule can run in constant time.

STM run-time manager in CHR. The effects of an STM transaction are reads and writes to shared memory. The STM run-time must guarantee that all reads and writes within a transaction happen logically at once. In case transactions are optimistically executed in parallel the STM run-time must take care of any potential read/write conflicts. The STM run-time must ensure that in case of conflicts at least one transaction can successfully commit its updates, whereas the other transaction is retried.

To accomplish this behavior, we use for each transaction a read log and a write log. Before we can commit the write log and actually update the memory cell, we first must validate that for each cell whose value is stored in the read log, the actual value is still the same.

In Figure 8, we specify the STM manager via CHR rules. It has been slightly simplified in this survey. Besides locations and values, we introduce an identifier for transactions `T`. The operation constraints are `read` and `write` and the protocol constraints are `validate, commit` and `rollback, retry`. The data constraint `CommitRight` acts as a token a committing transaction has to acquire in order to avoid concurrent writes. The constraint `validate` is issued at an end of the transaction if the `CommitRight` is available. Rules for rollback and retry of transactions are not shown here for space reasons.

Soundness and correctness. Our implementation guarantees atomicity, isolation and optimistic concurrency. It is therefore sound. It is correct: if a transaction commits successfully, the store reflects correctly all the reads/writes performed by that transaction.

10.2 Colored Petri Nets (CPN)

Petri Nets are diagrammatic formalism to describe and reason about concurrent processes. They consist of labeled *places* (◯) in which *tokens* (•) reside. Tokens can move along *arcs* passing through *transitions* (▭) from one place to another. A transition may have several incoming arcs and several outgoing arcs. A transition can only fire if all incoming arcs present a token. On firing, all incoming tokens will be removed and a token will be presented on each outgoing arc. *CPN* (Jensen, 1987) significantly generalize Petri Nets. Tokens are colored and places are typed by the colors they allow. Transitions can have conditions on tokens and equations that compute new tokens from old ones.

The paper (Betz, 2007) shows that (colored) Petri Nets can easily be embedded into CHR. When CPNs are translated to CHR, color tokens are encoded as numbers. Place

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

```
 % Execution phase ---
 % Read from write or read log, create read log otherwise
r1 : WLog(t,l,v1) \ Read(t,l,v2) <=> v1=v2.
r2 : RLog(t,l,v1) \ Read(t,l,v2) <=> v1=v2.
r3 : Cell(l,v1) \ Read(t,l,v2) <=> v1=v2, RLog(t,l,v1).

 % Write to write log, create write log otherwise
w1 : WLog(t,l,v1), Write(t,l,v2) <=> WLog(t,l,v2).
w2 : Write(t,l,v) <=> WLog(t,l,v).

 % Validation phase ---
 % Check and remove read log, rollback on read log conflict
v1 : Cell(l,v1), Validate(t) \ RLog(t,l,v2) <=> v1=v2 | True.
v2 : Cell(l,v1) \ Validate(t), RLog(t,l,v2) <=> v1=\=v2 | Rollback(t).

 % Start commit phase by acquiring CommitRight, otherwise rollback
s1 : CommitRight, Validate(t) <=> Commit(t).
s2 : Validate(t) <=> Rollback(t).

 % Commit phase ---
 % write update cells, then return CommitRight
c1 : Commit(t) \ Cell(l,v1), WLog(t,l,v2) <=> Cell(l,v2).
c2 : Commit(t) <=> CommitRight.
```

Fig. 8. STM run-time manager in CHR.

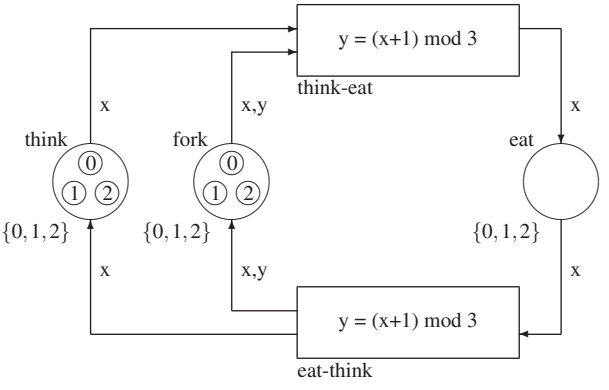

Fig. 9. Three dining philosophers problem as Colored Petri Net.

labels are mapped to CHR constraint symbols, tokens at a place to instances of CHR constraints, transitions and their arcs to simplification rules. Incoming arc places form the rule head, outgoing arc places form the rule body, and the transition conditions as well as equations form the rule guard.

Example 10.1

For simplicity, we consider the dining philosophers problem with just three philosophers as CPN in Figure 9. Each philosopher (and fork) corresponds to a colored token, given as a number from 0 to 2. Two philosophers x and y are neighboring if $y = (x+1) \bmod 3$. Places are think, eat and fork, transitions are eat–think and and think–eat.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

The CPN of Figure 9 translates into the following two CHR rules:

```
think_eat : think(X), fork(X), fork(Y) <=> Y =:= (X+1) mod n | eat(X).
eat_think : eat(X) <=> Y =:= (X+1) mod n | think(X), fork(X), fork(Y).
```

Soundness and completeness. For both classical and CPNs, these correctness theorems are proven for the translation into CHR.

10.3 Actor Model

In the Actor Model (Agha, 1986), one coordinates concurrent computations by message passing. Actors communicate by sending and receiving messages. Sending is a non-blocking asynchronous operation. Each sent message is placed in the actors mailbox (a message queue). Messages are processed via receive clauses which perform pattern matching and guard checks. Receive clauses are tried in sequential order. The receive operation is blocking. If none of the receive clauses applies, the actor suspend until a matching message is delivered. Receive clauses are typically restricted to a single-headed message pattern. That is, each receive pattern matches at most one message.

In Sulzmann *et al.* (2008), we extend the Actor Model with receive clauses allowing for multi-headed message patterns. Their semantics is inspired by their translation into CHR. We have implemented a prototype in Haskell `https://code.google.com/archive/p/haskellactor/`.

Example 10.2
In the Santa Clause problem, Santa sleeps until woken by either all of his nine reindeer or by three of his ten elves. If woken by the reindeer, he harnesses each of them to his sleigh, delivers toys and finally unharnesses them. If woken by three elves, he shows them into his study, consults with them on toys and finally shows them out. Here is a solution using the proposed multi-head extension:

```
santa sanActor =
  receive sanActor of
   Deer x1, Deer x2, ..., Deer x8, Deer x9 -> harness, deliver, unharness.
   Elf x1, Elf x2, Elf x3 -> enter_study, consult, leave_study.
```

This straightforward solution avoids the clumsiness of explicitly counting deers and elves in the mailbox. There is an obvious direct embedding of the matching receive clauses into CHR simplification rules.

Semantics of Actors with multi-headed message patterns. We study two possible semantics for this extension, inspired by the standard refined semantics of CHR:

- The *first-match semantics* provides a conservative extension of the semantics of single-headed receive clauses. This semantics guarantees monotonicity: Any successful match remains valid if further messages arrive in the actors mailbox.
- The *rule-order-match semantics* guarantees that rule patterns are executed in textual order. In this semantics, newly arrived messages can invalidate earlier match choices.

It will depend on the application in which semantics is the better choice.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

10.4 Join-calculus and join-patterns

In Join-Calculus (Fournet and Gonthier, 2002), concurrency is expressed via multi-headed declarative reaction rules that rewrite processes or events. The (left-hand side of a) rule is called *join-pattern*. They provide a high-level coordination of concurrent processes. The thesis (Lam, 2011b) extends join-patterns with guards and describes a prototype implementation in parallel CHR compiled to Haskell, see `http://code.haskell.org/parallel-join`.

Join-calculus with guarded join-patterns. A concurrent *process (or event)*, say `P`, has the form of a predicate. A *reaction rule (join-pattern)* rewrites processes. We introduce *guards* into these rules:

$$\text{Guarded Reaction Rule } P_1, \ldots P_n \text{ if Guard} \Rightarrow P'_1, \ldots P'_m$$

The join-calculus semantics is defined by a chemical abstract machine. This model specifies transformations using a chemical reaction metaphor. The chemical abstract machine can be embedded in CHR, see Chapter 6 in Frühwirth (2009).

Example 10.3
A print job is to be executed on any available printer where it fits. So print jobs have a size, and printers have a certain amount of free memory. This behavior is captured by the following guarded reaction rule:

```
ReadyPrinter(p,m), Job(j,s) if m>s => SendJob(p,j)
```

There is an obvious direct translation into CHR simplification rules.

Implementation and experimental results. Standard CHR goal-based lazy matching is a suitable model for computing the triggering of join-patterns with guards: Each process (CHR goal) essentially computes only its own rule head matches asynchronously and then proceeds immediately. We conducted experiments of our parallel join-calculus implementation with examples for common parallel programming problems. They show consistent speed-up as we increase the number of processors.

11 Discussion and future work

We now present common topics and issues that we have identified as a result of this survey and that lead to research questions for future work.

Syntactic fragments of CHR. The parallel and distributed semantics surveyed are concerned with expressive Turing-complete fragments of CHR. Their properties are summarized in Table 1. Except for the distributed semantics (CHRd and CHRe), they do not allow for terminating propagation rules. In the distributed semantics of CHRd and CHRe, one restricts rule heads to be sufficiently connected by shared variables, requiring direct-indexed and n-neighbor (star-shaped) rules, respectively. The former is no real restriction, the latter is.

Software implementations always presume Ground CHR (and so does CHRt). Hardware implementations in addition rely on *non-size-increasing rules* which are still Turing complete.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

Table 1. *Syntactic restrictions and properties of CHR parallel and distributed semantics*

CHR semantics	Syntactic restriction	Monotonicity soundness serializability
Abstract par.	Propagation rules do not terminate	Yes
Refined par.	No propagation rules	Yes
CHRmp	No propagation rules	Soundness for deletion-acyclic programs
CHRt	Ground data and operation constraints	Yes
CHRd	Direct-indexed rule heads	Yes for ground confluent programs?
CHRe	Ground star-shaped rule heads	For quiescent programs

Sometimes the notion of constraints is too abstract, and one differentiates between *data and operation constraints*. Operation constraints update data constraints. This dichotomy clarifies programs like Blocks World and UF, is essential in the semantics of CHRtand in the concurrency model of STM when encoded in CHR.

All example programs in the survey and in general many other sequential CHR programs can still be run in parallel without modification, since the syntactic restrictions are observed as they cover expressive subsets of CHR. However, changes are necessary if the program is not ground, for parallel execution, if the program contains propagation rules, and for distributed execution if the rule heads are not sufficiently connected. This need for program modifications weakens the promise of declarative parallelism, and therefore (semi-)automatic methods of program transformation should be investigated. Note that such transformations would be purely syntactical and do not require to come up with any scheduling for parallelism.

Propagation rules. Surprisingly, while propagation rules seem perfect for parallelization (because they do not remove any constraints), they are currently only supported in distributed CHRd and CHRe (see Table 1). (In the abstract parallel semantics, they are allowed, but do not terminate.) On the other hand, it seems possible to extend the refined parallel semantics with propagation rules, either using the *propagation history* of CHRe or the *occurrence check* approach of CHRd to avoid their trivial non-termination. The former seems to come with some implementation overhead, since the data structure needs to be updated in parallel. The latter approach does not work in all cases, but it could be applicable to set-based semantics like CHRmp. As for a third possibility, in the literature on optimizing CHR implementations, one can find program analyses that detect if propagation rules can be executed without any checks. Ground CHR is a good candidate for avoiding checks altogether, because constraints cannot be re-activated.

Semantics properties: Monotonicity, serializability and soundness. These properties have been proven for all parallel CHR semantics based on multi-sets, for distributed CHRe with the restriction to quiescent programs. Surprisingly, these properties do not hold in general for the set-based semantics of distributed CHRd and massively parallel CHRmp.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

The papers on CHRd do not fully investigate these properties, while CHRmp is sound for *deletion-acyclic programs*. Clearly, set-based semantics for CHR have to be studied more deeply. There seems to be a mismatch between their elegance of the concept and its actual behavior.

Program analysis. We should re-examine CHR program analysis for parallel and distributed CHR to see how they carry over. *Termination* corresponds to *quiescence* in the concurrent context. There is a vast literature on (non-)termination and complexity analysis of CHR programs. *Confluence* is an essential desirable property of sequential CHR programs. It already plays a role in parallel CHR for sound removal of transactions and seems trivial in exhaustively parallel CHRmp. Confluence seems strongly related to soundness and serializability properties of concurrent CHR semantics. Semi-automatic *completion* generates rules to make programs confluent. This method has been used in parallelizing the UF algorithm and can be used for translating away CHR transactions. When transactions are involved, confluence seems to avoid deadlocks. We also think that the property of *deletion-acyclicity* of CHRmp has a broader application in rule-based systems. It seems related to confluence and we think can be expressed as a termination problem.

Software and hardware implementations. All software implementations surveyed are available online for free download, the links have been given. The implementations cover parallel CHR, set-based CHRd and distributed CHRe as well as CoMingle. All implementations restrict themselves to the ground subset of CHR. A full-fledged widely used stable implementation of parallel CHR is still missing. It could serve as a basis to foster further research and applications, as does the K.U. Leuven platform for sequential CHR. With CoMingle, the situation seems better in the case of distributed CHR. In any case, more evidence in the form of experimental results is needed to further confirm the promise of declarative concurrency made by CHR.

Models of concurrency in CHR. Embedding models of concurrency in CHR is promising for understanding, analyzing and extending models, but still in its infancy. It is appealing because of the *lingua franca* argument for CHR: Different embeddings can be compared on its common basis and fertilize each other. Conversely, the striking similarity of the some models when encoded in CHR leads one to speculate about a generic concurrency model that is a suitable fragment of CHR which could then be mapped to many existing models, yielding a truly unified approach.

12 Conclusions

We have given an exhaustive survey of abstract and more refined semantics for parallel CHR as well as distributed CHR. Most of them have been proven correct. These semantics come with several implementations in both software and hardware. All software implementations are available online for free download. We presented non-trivial classical example programs and promising experimental results showing parallel speed-up. Last but not the least, we reviewed concurrency models that have been encoded in CHR to get a better understanding of them and sometimes to extend them. Most of these embeddings have been proven correct, i.e., sound and complete. Some embeddings are available online.

https://doi.org/10.1017/S1471068418000078 Published online by Cambridge University Press

In the discussion, we identified the following main topics for future work: Including propagation rules into the parallel semantics and providing program transformations into the expressive syntactic fragments for distributed CHR, investigate set-based semantics and the deletion-acyclic programs, provide a full-fledged implementation of parallel CHR, apply the wealth of existing program analyses for sequential CHR to distributed and parallel CHR programs and the embedding of concurrency models, and explore similarities of the concurrency models embedded in CHR as lingua franca to come up with unified models.

On a more general level, it should be investigated how the research surveyed here carries over to related languages like constraint logic programming ones and the other rule-based approaches that have been embedded in CHR. Overall, the CHR research surveyed here should be related to more mainstream research in concurrency, parallelism and distribution.

Acknowledgements

We thank the anonymous referees for their helpful, detailed and demanding suggestions on how to improve this survey.

References

ABDENNADHER, S. AND FRÜHWIRTH, T. 1998. On completion of Constraint Handling Rules. In *Proc. International Conference on Principles and Practice of Constraint Programming*, M. J. Maher and J.-F. Puget, Eds. Lecture Notes in Computer Science, vol. 1520. Springer, 25–39.

ABDENNADHER, S. AND FRÜHWIRTH, T. 1999. Operational equivalence of CHR programs and constraints. In *Proc. International Conference on Principles and Practice of Constraint Programming*, J. Jaffar, Ed. Lecture Notes in Computer Science, vol. 1713. Springer, 43–57.

ABDENNADHER, S. AND FRÜHWIRTH, T. 2004. Integration and optimization of rule-based constraint solvers. In *Proc. International Symposium on Logic-Based Program Synthesis and Transformation*, M. Bruynooghe, Ed. Lecture Notes in Computer Science, vol. 3018. Springer, 198–213.

ABDENNADHER, S., FRÜHWIRTH, T. AND MEUSS, H. 1999. Confluence and semantics of constraint simplification rules. *Constraints 4,* 2, 133–165.

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.

BETZ, H. 2007. Relating coloured Petri nets to Constraint Handling Rules. In *Proc. 4th Workshop on Constraint Handling Rules*, 33–47.

BETZ, H. 2014. *A Unified Analytical Foundation for Constraint Handling Rules*. BoD–Books on Demand.

BETZ, H., RAISER, F. AND FRÜHWIRTH, T. 2010. A complete and terminating execution model for constraint handling rules. *Theory and Practice of Logic Programming 10*, 597–610.

CERVESATO, I., LAM, E. S. L. AND ELGAZAR, A. 2016. *Choreographic Compilation of Decentralized Comprehension Patterns*. Springer International Publishing, Cham, 113–129.

DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M. AND HOLZBAUR, C. 2004. The refined operational semantics of Constraint Handling Rules. In *Proc. International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer, 90–104.

FOURNET, C. AND GONTHIER, G. 2002. *The Join Calculus: A Language for Distributed Mobile Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 268–332.

FRÜHWIRTH, T. 2005a. Parallelizing union-find in Constraint Handling Rules using confluence. In *Proc. International Conference on Logic Programming*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 113–127.

FRÜHWIRTH, T. 2005b. Specialization of concurrent guarded multi-set transformation rules. In *Proc. International Symposium on Logic-Based Program Synthesis and Transformation*, S. Etalle, Ed. Lecture Notes in Computer Science, vol. 3573. Springer, 133–148.

FRÜHWIRTH, T. 2006. Deriving linear-time algorithms from union-find in CHR. In *CHR '06*, T. Schrijvers and T. Frühwirth, Ed. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452, 49–60.

FRÜHWIRTH, T. 2009. *Constraint Handling Rules (Monography)*. Cambridge University Press.

FRÜHWIRTH, T. 2015. Constraint handling rules – what else? In *Rule Technologies: Foundations, Tools, and Applications*. N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke and D. Roman, Eds. Springer International Publishing, 13–34.

FRÜHWIRTH, T. 2016. *The CHR Web Site*. Accessed May 2018 URL: `www.constraint-handling-rules.org`. Ulm University.

FRÜHWIRTH, T. AND HOLZBAUR, C. 2003. Source-to-source transformation for a class of expressive rules. In *Proc. Joint Conf. Declarative Programming APPIA-GULP-PRODE*, F. Buccafurri, Ed. 386–397.

FRÜHWIRTH, T. AND RAISER, F., Ed. 2011. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand.

GABBRIELLI, M., MEO, M. C., TACCHELLA, P. AND WIKLICKY, H. 2013. Unfolding for CHR programs. *Theory and Practice of Logic Programming*, *15*, 3, 1–48.

GOLDBERG, A. V. AND TARJAN, R. E. 1988. A new approach to the maximum-flow problem. *J. ACM 35,* 4, 921–940.

GUERRAOUI, R. AND KAPALKA, M. 2008. On the correctness of transactional memory. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 175–184.

HOLZBAUR, C., GARCÍA DE LA BANDA, M., STUCKEY, P. J. AND DUCK, G. J. 2005. Optimizing compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming* 5, 4–5, 503–531.

JENSEN, K. 1987. *Coloured Petri Nets*. Springer, Berlin, Heidelberg, 248–299.

LAM, E. S. 2018. Concurrent CHR, chapter 5. In *Constraint Handling Rules: Compilation, Execution, and Analysis*, T. Frühwirth and F. Raiser, Eds. Books on Demand, 121–155.

LAM, E. S. AND CERVESATO, I. 2013. Decentralized execution of constraint handling rules for ensembles. In *Proc. 15th Symposium on Principles and Practice of Declarative Programming*. ACM, 205–216.

LAM, E. S. AND SULZMANN, M. 2007. A concurrent constraint handling rules semantics and its implementation with software transactional memory. In *Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. ACM Press.

LAM, E. S. AND SULZMANN, M. 2009. Concurrent goal-based execution of constraint handling rules. *Theory and Practice of Logic Programming 11*, 841–879.

LAM, E. S. L. 2011. Parallel execution of constraint handling rules – Theory, implementation and application. Ph.D. thesis, School of Computing, Department of Computing Science, National University of Singapore.

LAM, E. S. L., CERVESATO, I. AND FATIMA, N. 2015. Comingle: Distributed logic programming for decentralized mobile ensembles. In *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015*, 51–66.

MEISTER, M. 2007. Concurrency of the preflow-push algorithm in constraint handling rules. In *Proc. 12th International Workshop on Constraint Solving and Constraint Logic Programming*, 160–169.

MEISTER, M. AND FRÜHWIRTH, T. 2007. Reconstructing almost-linear tree equation solving algorithms in CHR. In *Proc. Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming*, 123.

RAISER, F., BETZ, H. AND FRÜHWIRTH, T. 2009. Equivalence of CHR states revisited. In *Proc. Constraint Handling Rules*, F. Raiser and J. Sneyers, Eds. K.U. Leuven, Dept. Comp. Sc., Technical report CW 555, 33–48.

RAISER, F. AND FRÜHWIRTH, T. 2010. Exhaustive parallel rewriting with multiple removals. In *WLP '10*, S. Abdennadher, Ed.

SARNA-STAROSTA, B. 2008. *Constraint-Based Analysis of Security Properties*. VDM Verlag, Saarbrücken, Germany.

SARNA-STAROSTA, B. AND RAMAKRISHNAN, C. 2007. Compiling constraint handling rules for efficient tabled evaluation. In *Proc. 9th International Symposium on Practical Aspects of Declarative Languages*, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 4354. Springer, 170–184.

SARNA-STAROSTA, B., STIREWALT, R. E. K. AND DILLON, L. K. 2007. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. *International Journal of Software Engineering and Knowledge Engineering 17,* 2, 207–230.

SCHRIJVERS, T. AND SULZMANN, M. 2008. Transactions in constraint handling rules. In *Proc. 24th International Conference on Logic Programming*. Lecture Notes in Computer Science, vol. 5366. Springer, 516–530.

SHAVIT, N. AND TOUITOU, D. 1997. Software transactional memory. *Distributed Computing 10,* 2, 99–116.

SNEYERS, J. 2008. Turing-complete subclasses of CHR. In Proc. *24th International Conference on Logic Programming.* Lecture Notes in Computer Science, vol. 5366. Springer, 759–763.

SNEYERS, J., SCHRIJVERS, T. AND DEMOEN, B. 2009. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS 31,* 2, 3–42.

SNEYERS, J., VAN WEERT, P., SCHRIJVERS, T. AND DE KONINCK, L. 2010. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *TPLP 10,* 1, 1–47.

SULZMANN, M. AND CHU, D. H. 2008. A rule-based specification of Software Transactional Memory. In *LOPSTR '08, Pre-proceedings*, M. Hanus, Ed.

SULZMANN, M. AND LAM, E. S. 2007. Compiling constraint handling rules with lazy and concurrent search techniques. In *Proc. 4th Workshop on Constraint Handling Rules*, 139–149.

SULZMANN, M. AND LAM, E. S. 2008. Parallel execution of multi-set constraint rewrite rules. In *Proc. 10thInternational Conference on Principles of Practical Declarative Programming*, S. Antoy, Ed. ACM Press, 20–31.

SULZMANN, M., LAM, E. S. AND VAN WEERT, P. 2008. Actors with multi-headed message receive patterns. In *Proc. 10th International Conference on Coordination Models and Languages*, D. Lea and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 5052. Springer, 315–330.

TARJAN, R. E. AND LEEUWEN, J. V. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM 31,* 2, 245–281.

TRIOSSI, A. 2011. Hardware execution of constraint handling rules. PhD Thesis, Universita Ca Foscari di Venezia.

TRIOSSI, A., ORLANDO, S., RAFFAETÀ, A. AND FRÜHWIRTH, T. 2012. Compiling chr to parallel hardware. In *Proc. 14th Symposium on Principles and Practice of Declarative Programming*. ACM, 173–184.

VAN WEERT, P. 2010. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering 22,* 11, 1521–1534.

ZAKI, A., FRÜHWIRTH, T. AND GELLER, I. 2012. Parallel execution of constraint handling rules on a graphical processing unit. In *CHR '12*, J. Sneyers and T. Frühwirth, Eds. K.U. Leuven, Dept. Comp. Sc., Technical report CW 624, 82–90.