

Transactions in Constraint Handling Rules

Tom Schrijvers and Martin Sulzmann

¹ Department of Computer Science, K.U.Leuven, Belgium
e-mail: tom.schrijvers@cs.kuleuven.be

² ITU, Copenhagen, Denmark
e-mail: martin.sulzmann@gmail.com

Abstract. CHR is a highly concurrent language, and yet it is by no means a trivial task to write correct concurrent CHR programs. We propose a new semantics for CHR, which allows specifying and reasoning about *transactions*. Transactions alleviate the complexity of writing concurrent programs by offering entire derivations to run atomically and in isolation.

We derive several program transformations based on our semantics that transform particular classes of transitional CHR programs to non-transactional ones. These transformations are useful because they obviate a general purpose transaction manager, and may lift unnecessary sequentialization present in the transaction AI semantics.

1 Introduction

Constraint Handling Rules (CHR) [5] is a concurrent committed-choice constraint logic programming language. Each CHR rewrite rule specifies an atomic and isolated transformation step (rewriting) among a multi-set of constraints. Although this greatly facilitates writing concurrent programs, often we wish for entire derivations of CHR rewrite steps to be executed atomically and in isolation. For this purpose we propose CHR^{*}, an extension of CHR where the user can group together sets of goals (constraints) in a transaction. Like database transactions, atomically executed transactions provide an “all-or-nothing” guarantee. The effects of the transaction are visible either in their entirety or the transaction has no effect. Further, transactions are executed in isolation from other transactions. That is, no intermediate state is observable by another transaction.

The efficient implementation of transactions is quite challenging: the amount of concurrency should be maximized with a minimum of synchronization overhead. For this purpose, non-interfering transactions should be identified for concurrent scheduling on different processors or processor cores. The current state of the art for dealing with concurrent transactions is *optimistic concurrency control*. In essence, each transaction is executed under the optimistic assumption that concurrent transactions do not interfere with each other. Each transaction’s updates are logged and only committed once the transaction is fully executed, and only no update conflicts with other transactions. In theory, this method supports a high-level of concurrency. Unfortunately, in practice many “false” conflicts may

arise in case the objects protected by transactions are subject to a high-level of contention. The consequence is a significantly lower level of concurrency. This is a serious problem for the practical acceptance of transactions in the programming language setting.

Our novel idea is that instead of being concerned with a generic and efficient execution scheme for CHR^* which enables a high level of concurrency, we investigate meaning-preserving transformation methods of CHR^* to plain CHR programs. The advantages of this approach are that there is no overhead caused by a concurrency control scheme for transactions and "false" conflicts are completely avoided. Furthermore, resulting plain CHR programs enjoy a high-level of concurrency. In some cases, we can apply domain-specific optimizations to even further boost the level of concurrency. The correctness of our transformation methods is based on well-defined criteria and formal reasoning techniques. Our criteria and techniques cover a certain, we claim significant, class of CHR^* programs.

In summary, we make the following contributions:

- We introduce CHR^* (pronounce: Atom CHR) an extension of CHR with atomic transactions and develop the meta-theory for such a calculus (Section 4). We demonstrate the usefulness of CHR^* via a number of examples.
- We devise an execution scheme for CHR^* where transactions must be executed sequentially to guarantee atomicity and isolation but plain CHR derivations can be executed concurrently whenever possible (Section 5).
- Our approach to unlock concurrency in CHR^* works by means of transformation to plain CHR. Specifically, we consider transformation methods (Section 6) which cover
 - bounded transactions,
 - (partially) confluent transactions, and
 - domain-specific transaction synchronization and recovery.

Section 2 provides an overview of our work. Section 3 gives background material on the CHR language. CHR follows Prolog syntax conventions, where identifiers starting with a lower case letter indicate predicates and function symbols, and identifiers starting with upper case. We will stick to this convention throughout the paper. Discussion of related work is deferred until Section 7. Section 8 concludes.

Additional material such as examples are given in an accompanying technical report [13].

2 Overview

In this section, we present two examples to motivate CHR^* and our transformation methods. In the first example, we see a bounded transaction, i.e. whose number of derivation steps is bounded. In the second example, transactions are unbounded.

```

    balance(acc1,2000) ∧ balance(acc2,0) ∧ balance(acc3,1000)
  ∧ transfer(acc1,acc2,1000)
  ∧ transfer(acc1,acc3,1500)
  ↦ (unfold transfer x2)
    balance(acc1,2000) ∧ balance(acc2,0) ∧ balance(acc3,1000)
  ∧ withdraw(acc1,1000) ∧ deposit(acc2,1000)
  ∧ withdraw(acc1,1500) ∧ deposit(acc3,1500)
  ↦ (deposit(acc2,1000))
    balance(acc1,2000) ∧ balance(acc2,1000) ∧ balance(acc3,1000)
  ∧ withdraw(acc1,1000)
  ∧ withdraw(acc1,1500) ∧ deposit(acc3,1500)
  ↦ (deposit(acc3,1500))
    balance(acc1,500) ∧ balance(acc2,1000) ∧ balance(acc3,1000)
  ∧ withdraw(acc1,1000)
  ∧ deposit(acc3,1500)
  ↦ (deposit(acc3,1500))
    balance(acc1,500) ∧ balance(acc2,1000) ∧ balance(acc3,2500)
  ∧ withdraw(acc1,1000)

```

Table 1. Non-atomic transfer

2.1 Bounded Transaction: Bank Transfer

Consider these (plain) CHR rules for updating a bank account:

```

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

```

The `balance/2` constraint is a data constraint, and the `deposit/2` and `withdraw/2` constraints are operation constraints. The guard ensures that a withdraw is only possible if the amount in the account is sufficient. In the concurrent CHR semantics, rules can be applied simultaneously, as long as they operate on non-overlapping parts of the constraint store. Simultaneous deposit or withdrawal from distinct accounts is therefore possible.

The transfer constraint/rule combines deposit and withdraw among two accounts.

```

transfer(Acc1,Acc2,Amount) <=>
    withdraw(Acc1,Amount) ∧ deposit(Acc2,Amount).

```

Suppose, we wish to perform two transfers where three accounts are involved.

```

    balance(acc1,2000) ∧ balance(acc2,0) ∧ balance(acc3,1000)
  ∧ transfer(acc1,acc2,1000)
  ∧ transfer(acc1,acc3,1500)

```

Table 1 shows a possible derivation. We simulate concurrency using an interleaving semantics. The point to note is that we cannot execute `withdraw(acc1, 1000)` because of insufficient funds. The result is a "non-atomic" execution of the first transfer operations. We only manage to deposit but fail to withdraw. This is clearly not the desired behavior of a transfer.

In CHR^{\star} , such problems can be avoided by guarding each `transfer` constraint with an `atomic()` wrapper. In general, `atomic()` can be wrapped around any constraint C (possibly consisting of a conjunction of constraints). We refer to `atomic(C)` as a *transaction*. The CHR^{\star} semantics guarantees that transactions are executed atomically and in isolation from other transactions. Informally, this means that all constraints (and also those occurring in subderivations) are either executed exhaustively or not at all. Any store updates are only visible once the transaction is completed.

Next, we consider transformation of the above CHR^{\star} program to plain CHR. In this example, derivation steps within a transaction are bounded. By performing a simple unfolding of CHR, we can replace the atomic transfer rule by the following single rule.

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

Immediately, this multi-headed rule expresses the fact that an atomic transfer requires exclusive access to both accounts involved. Section 6.1 contains the details of the transformation method for bounded transactions.

2.2 Unbounded Transaction: Shared Linked List

In this example, we consider linked lists of distinct elements (numbers) in increasing order. A list is made up of four kinds of data constraints:

- We write `node(X,P,Q)` to represent a node with value X at location (cfr. pointer address) P whose tail is at location Q .
- We write `nil(P)` to represent an empty list at location P .
- We write `found(X)` and `notfound(X)` to indicate that either a value X is part of the list or not.

Two operation constraints inspect and modify a list:

- The `find(X,P)` operation searches for the element X starting from the location P .
- Operation `insert(X,P)` adds the element X to the linked list at the proper position (if not already present).

For brevity's sake, we assume the existence of the primitive `fresh` for the generation of fresh locations¹. The program is:

¹ This could be encoded via additional CHR rules

```

f1 @ node(X,P,Next) \ find(X,P) <=> found(X).
f2 @ nil(P) \ find(Y,P) <=> notfound(Y).
f3 @ node(X,P,Next) \ find(Y,P) <=> X < Y | find(Y,Next).
f4 @ node(X,P,Next) \ find(Y,P) <=> X > Y | notfound(Y).

i1 @ node(X,P,Next) \ insert(X,P) <=> true.
i2 @ node(X,P,Next) \ insert(Y,P) <=> X < Y | insert(Y,Next).
i3 @ node(X,P,Next), insert(Y,P) <=> X > Y |
    fresh(NewP) ^ node(Y,P,NewP) ^ node(X,NewP,Next).
i4 @ nil(P), insert(X,P) <=>
    fresh(NewP) ^ node(X,P,NewP) ^ nil(NewP).

```

In CHR^{*}, we can then specify the atomic and isolated execution of operations, for example (we omit the data store holding the linked list for simplicity)

```
atomic(find(50,root)) ^ atomic(insert(2,root))    (***)
```

Transactions are unbounded because operations traverse a dynamic data structure. It is a well-known fact that in case the linked list is subject to a high-level of contention, an optimistic concurrency control scheme for transactions does not exhibit a high-level of concurrency due to many "false" conflicts. Details are in Section 5.1.

To safely remove `atomic()` wrappers (and thus avoiding "false" conflicts) we establish criteria which guarantee that the resulting program still behaves atomically and does not violate isolation. One of the key criteria is partial confluence of the primitive operations (i.e. plain CHR rules) out of which transactions are composed of. Partial confluence means that non-joinable critical pairs are either ruled out because they disobey an invariant (i.e. they are observably confluent [4]), or non-joinability can be explained as a specific serial (in essence indeterministic) execution of the (atomic) operations. The above linked list rules are partially confluent which therefore guarantees that isolation is not violated once we remove the `atomic()` wrappers in (***). Details are in Section 6 where we also discuss more complex transactions and the impact of additional operations such as delete.

3 Preliminaries: Concurrent CHR

We assume that the reader is already familiar with CHR ([6]), and restrict ourselves here to the conventions used in this paper.

The Concurrent CHR language For the purpose of this paper, we consider only a subset the CHR language, with restrictions on two accounts.

Firstly, programs do not involve Prolog-style logical variables and built-in constraint solvers over these logical variables. In other words, all CHR constraints range over ground terms.

Syntactic Categories			
A, B, C	zero or more constraints	O	one operation constraint
D	one data constraint	E, F	zero or more operation constraints
S, T	zero or more data constraints	G	(Boolean) guard constraint
Basic Rule Application			
	$(C \Leftrightarrow G B)$ is a fresh rule variant		
(SIMPLIFY)	$C\theta \equiv C' \quad CT \models G\theta$		$C' \mapsto B\theta$
	$(C \Rightarrow G B)$ is a fresh rule variant		
(PROPAGATE)	$C\theta \equiv C' \quad CT \models G\theta$		$C' \mapsto C' \wedge B\theta$
Concurrent CHR			
(MONOTONICITY)	$A \mapsto B$		(WEAKPAR) $\frac{A_1 \mapsto A_2 \quad B_1 \mapsto B_2}{A_1 \wedge B_1 \mapsto A_2 \wedge B_2}$
	$A \wedge C \mapsto B \wedge C$		
(STRONGPAR)	$A_1 \wedge C \mapsto A_2 \wedge C$	$B_1 \wedge C \mapsto B_2 \wedge C$	$A_1 \wedge B_1 \wedge C \mapsto A_2 \wedge B_2 \wedge C$

Fig. 1. Concurrent CHR

Secondly, CHR constraint symbols C are partitioned into two classes \mathcal{C}_D and \mathcal{C}_O . The former are *data* constraints and the latter *operation* constraints, according to the terminology of [14]. See Fig. 1 for the notational conventions used in this paper. Moreover, the head of each CHR rule must consist of exactly one operation constraint and zero or more data constraints, i.e. it is of the form O, D_1, \dots, D_n with $n \geq 0$.

Neither of these restrictions is very demanding. In fact, Sneyers et al [15] show that this fragment of the CHR language is Turing-complete: their RAM machine simulator works on ground constraints and the program counter constraint can be considered the operation constraint while the other constraints are data constraints.

Operational Semantics Frühwirth [7] proposes a (non-transactional) concurrent operational semantics for CHR (bottom of Fig. 1) on top of the basic rule applications (middle of Fig. 1). Rule MONOTONICITY encodes the monotonicity property of CHR, which is also part of the sequential CHR semantics. Rule WEAKPAR is the traditional compositionality property, which allows for derivations to run fully isolated and their results to be merged. The rule STRONGPAR is proposed by Frühwirth in [7]. It allows for a stronger form of concurrency, as both subderivations may share the same unmodified context E .

A CHR derivation starting from constraints C_0 yields after exhaustive application of the rules of Fig. 1 the constraints C_n . Such a derivation is denoted as $C_0 \mapsto^* C_n$. As the CHR semantics is nondeterministic there may be many

derivations, with different final constraints C_n , starting from the same C_0 . We denote $\mathcal{S}[[P]](C_0)$ the set $\{C_n \mid C_0 \mapsto^* C_n\}$ wrt. CHR program P , i.e. the set of different possible final constraints.

4 The CHR[★] Language

We propose CHR[★] (pronounce: Atom CHR), a new extension of CHR with atomic transactions. An atomic transaction is denoted as `atomic`(C) where C is a conjunction of CHR constraints. Atomic transactions may appear in queries and rule bodies, in addition to ordinary (non-transactional) constraints.

The semantics of CHR[★] is an extension of the ordinary CHR semantics of Figure 1. It requires only one more rule:

$$\boxed{\text{(ATOMIC)} \frac{T \wedge S_i \wedge C_i \mapsto^* T \wedge S'_i}{T \wedge \bigwedge_{i=1}^n S_i \wedge \bigwedge_{i=1}^n \text{atomic}(C_i) \mapsto T \wedge \bigwedge_{i=1}^n S'_i}}$$

The ATOMIC rule is quite general in nature. It defines a derivation step that runs any number of atomic transactions `atomic`(C_i) in parallel.

Each transaction is *isolated* from the others: the parallel step considers the separate evaluation of each C_i in isolation from the other C_j . The different transactions only share the common data, in our case constraints, $T \wedge \bigwedge_i S_i$. However, note that non of the transactions should observe updates S'_i from the other transactions to these common constraints.

Moreover, each transaction should run its full course, and not get stuck, i.e. no operation constraints should be left. This ensures that the context $T \wedge \bigwedge_i S_i$ provides sufficient data for the atomic derivation to have its intended effect. Without this condition, we could assume the context $T \wedge \bigwedge_i S_i$ to be empty, and thus lift the `atomic` wrapper and subsequently run the atomic derivation in a non-atomic manner.

The notation $\mathcal{S}^{\star}[[P]](C)$ for CHR[★] has the same meaning as $\mathcal{S}[[P]](C)$ for plain CHR.

In Section 2, we have seen several examples showing the usefulness of CHR[★]. We refer to [13] for a more extensive collection of examples.

4.1 Properties of CHR[★]

An important property of transactions in general is *serializability*: for each parallel execution of transactions there is a sequential execution with the same outcome [18]. This serializability property also holds for CHR[★].

Theorem 1 (Serializability). *For each ATOMIC derivation step $C_1 \mapsto C_2$ with n concurrent transactions, there is a corresponding derivation $C_1 \mapsto^n C_2$ of n consecutive ATOMIC steps each with only one transaction.*

The proof is straightforward, and the n transactions can be be serialized in any order. A corollary of serializability is that is that in the worst case of any CHR[★] derivation involves no concurrency at all.

While CHR^{\star} is a syntactic and semantic extension of CHR, the `atomic` keyword in fact restricts the possible derivations for a query with respect to concurrent CHR. For this purpose we define the notion of erasure, which drops the `atomic` keyword from a syntactic object:

Definition 1 (Erasure). *The erasure from a syntactic CHR object o is denoted as:*

$$\begin{aligned} \epsilon(C) &= C && (C \text{ doesn't contain } \text{atomic}) \\ \epsilon(C_1 \wedge C_2) &= \epsilon(C_1) \wedge \epsilon(C_2) \\ \epsilon(\text{atomic}(C)) &= C \\ \epsilon(C_1 \langle \Rightarrow \rangle C_2) &= C_1 \langle \Rightarrow \rangle \epsilon(C_2) \\ \epsilon(C_1 \Rightarrow C_2) &= C_1 \Rightarrow \epsilon(C_2) \\ \epsilon(C_1 \mapsto C_2) &= \epsilon(C_1) \mapsto \epsilon(C_2) \end{aligned}$$

We say that C is fully erased iff $\epsilon(C) \equiv C$.

Theorem 2 (Soundness under Erasure). *If $C_0 \mapsto^* C_n$ is an CHR^{\star} derivation wrt. program P , then $\epsilon(C_0 \mapsto^* C_n)$ is a plain CHR derivation wrt. $\epsilon(P)$.*

Corollary 1. *The set of resulting constraints of an CHR^{\star} program P is a subset of its erased form: $\forall P, \forall C. \mathcal{S}^{\star}[P](C) \subseteq \mathcal{S}[\epsilon(P)](\epsilon(C))$.*

Note that vice versa, i.e. by adding the `atomic` keyword to a concurrent CHR derivation, we do not necessarily obtain a valid CHR^{\star} derivation. In Section 6 we will see cases where it is valid to do so. Of course, for fully erased programs both CHR^{\star} and CHR yield the same result:

Theorem 3 (Completeness of Fully Erased Programs). *If CHR^{\star} program P and constraints C_0 are fully erased, then $\mathcal{S}^{\star}[P](C_0) = \mathcal{S}[P](C_0)$.*

5 CHR^{\star} Execution Schemes

We first sketch a possible optimistic concurrency control scheme for CHR^{\star} . The common problem of all optimistic concurrency methods is that the read logs of long running transactions may cause “false” conflicts with the write logs of other transactions. These conflicts lead to a roll back and therefore decreases the level of concurrency significantly. We therefore argue for a simple CHR^{\star} execution scheme where transactions are executed sequentially but which allows for concurrent execution of CHR derivation steps whenever possible. To unlock concurrency in CHR^{\star} programs, we investigate several transformation schemes from CHR^{\star} to CHR programs in the upcoming Section 6.

5.1 Optimistic Concurrency in CHR^{\star} and its Problem

Optimistic Concurrency in CHR^{\star} We apply the principle of optimistic concurrency control to CHR^{\star} as follows. Each transaction is executed optimistically, keeping a log of all reads and writes to the shared store. At the end of a transaction, we check for conflicts with the read/write logs of other transactions. For example, a conflict arises if the read log of a transaction overlaps with the write

log of another transaction. In this case, it is the transaction manager’s duty to resolve the conflict by for example rolling back one transaction² and letting the other transaction commit its write log.

The Problem In case shared data objects are subject to a high level of contention, optimistic concurrency control of atomic transactions degenerates to a sequential execution scheme.

Example 1. Let’s consider the earlier shared linked list example from Section 2.2. Suppose, we execute the two transactions $\text{atomic}(\text{find}(50, n_1)) \wedge \text{atomic}(\text{insert}(2, n_1))$ on the the shared store

$\text{node}(1, n_1, n_3) \wedge \text{node}(3, n_3, n_4) \wedge \dots \wedge \text{node}(50, n_{50}, n_{51}) \wedge \text{nil}(n_{51})$

The first transaction will search through the entire list whereas the second transaction will insert a new node containing 2 in between the first and second node. Both transactions “overlap”. That is, the read log of the first transaction conflicts with the write log of the second transaction. When searching for (the last) element 50, we will read the entire list, among others $\text{node}(1, n_1, n_3)$. However, the second transaction will re-write this constraint to $\text{node}(1, n_1, n_2)$, and adds $\text{node}(2, n_2, n_3)$. This is a conflict between the read of the first transaction and the write of the second. Assuming that the second transaction commits first, we are forced to roll back the first transaction. In other words, the sequential execution of both transactions is enforced. This is unfortunate because the read/write conflict is a “false” conflict. The result of the find transaction is independent of the insertion of node 2. Hence, we would expect that both transactions can be executed concurrently.

5.2 A Simple CHR^{*} Execution Scheme

CONCURRENTCHR	$\frac{C \mapsto_{CHR} C'}{C, S \mapsto_{SeqACHR} C', S}$
SEQUENTIALATOMIC	$\frac{\forall C. C \wedge C_1 \wedge C_2 \mapsto_{SeqACHR}^* C \wedge C_3}{C_1, \text{atomic}(C_2) \wedge S \mapsto_{SeqACHR} C_3, S}$
COLLECTATOMIC	$C_1 \wedge \text{atomic}(C_2), S \mapsto_{SeqACHR} C_1, \text{atomic}(C_2) \wedge S$

Fig. 2. Sequential CHR^{*} and Concurrent CHR Calculus

We consider a simple, sequential execution scheme for CHR^{*} programs but which allows for concurrent execution of CHR derivation steps whenever possible.

² i.e. we restart the transaction with an empty log.

Hence, we adopt the derivation rules of the concurrent CHR calculus, but we enforce that atomic transactions must be executed sequentially. Thus, we trivially guarantee the atomic and isolated execution of transactions. The semantics of such a restricted CHR^{\star} calculus is given in Fig. 2. We define the rewrite relation in this calculus in terms of judgments $C, S \mapsto_{\text{SeqACHR}} C', S'$ where C, C' are as before but S, S' refer to conjunctions of transactions. Rule `SEQUENTIALATOMIC` schedules a single transaction for (sequential) execution. Like in the case of rule `ATOMIC`, we need to ensure that operations within transactions are fully executed and do not get stuck. Rule `COLLECTATOMIC` collects all newly derived transactions. Rule `CONCURRENTCHR` switches to the concurrent CHR system.

In a concrete implementation, we could for example use a stack to systematically execute transactions. However, we wish to obtain a calculus in which we can represent all possible serializations of atomic transactions.

Theorem 4 (Equivalence). *We have that $C \mapsto C'$ is derivable in the CHR^{\star} calculus from Section 4 iff $C, \text{True} \mapsto_{\text{SeqACHR}} C'', S$ is derivable where $C' = C'' \wedge S$.*

6 From CHR^{\star} to CHR by Transformation

Our goal is to erase all (or as many as possible) `atomic()` wrappers from CHR^{\star} programs such that we can execute them in the concurrent CHR fragment. Of course, simply erasing the `atomic()` wrappers is not sound: the atomicity and isolation properties are easily lost.

Hence, in order to preserve atomicity and isolation of transactions under erasure, we not only erase the `atomic()` wrappers, but also perform other transformation steps on the program. These proposed transformations are detailed in the following subsections.

Some of our proposed transformations are applicable statically (off-line), others are only valid if the execution environment satisfies certain conditions. We could either check for these conditions dynamically, that is, we apply transformations on-line, or the programmer guarantees that under all execution paths the conditions are met. The techniques employed to carry out and verify the transformations range from simple unfolding methods to more sophisticated confluence analyses.

As a guideline for the correctness of transformations, we have the following generic criterion.

Definition 2 (Erasure Correctness Criterion). *A plain CHR program P' is correct erased form of an CHR^{\star} program P , iff $\forall C. \mathcal{S}^{\star}[[P]](C) = \mathcal{S}[[P']](\epsilon(C))$.*

As we observed earlier in Theorem 2, usually $\epsilon(P)$ does not satisfy the Erasure Correctness Criterion (ECC) as we only have that $\mathcal{S}^{\star}[[P]](C) \subseteq \mathcal{S}[[\epsilon(P)]](\epsilon(C))$. Hence, we study more involved transformations and conditions below.

Some transformations do not directly establish the ECC: they only eliminate particular atomic transaction wrappers. In such cases a more compositional approach is interesting: several transformations are combined to eliminate all transactions. Hence, we will usually prove a weaker criterion.

Definition 3 (Partial Erasure Correctness Criterion). An CHR^\star program P' is correct partially erased form of an CHR^\star program P , iff

$$\forall C. \mathcal{S}^\star \llbracket P \rrbracket (C) = \mathcal{S}^\star \llbracket P' \rrbracket (\epsilon(C))$$

By combining multiple transformations, we may end up with a fully erased program. Hence, from Theorem 3 and the Partial Erasure Correctness of each individual transformation, it then follows that the ECC holds.

For instance, a rather trivial form of erasure is erasure of operation-free atomic transactions. In other words, let $\epsilon'(o)$ be defined as $\epsilon(o)$, except that only $\epsilon'(\text{atomic}(S)) = S$ and not the more general $\epsilon'(\text{atomic}(C)) = C$. For this operation-free erasure we get trivially:

Theorem 5 (Correctness of Operation-Free Erasure).

$$\forall P. \forall C. \mathcal{S}^\star \llbracket P \rrbracket (C) = \mathcal{S}^\star \llbracket \epsilon'(P) \rrbracket (\epsilon'(C))$$

While the transformation is trivial, it nevertheless is a useful building block for composite transformations. The more interesting transformations are discussed below.

6.1 Bounded Transactions as Multi-Headed CHR Rules

We formalize the observations from the earlier Section 2.1 which observed that bounded transactions can be replaced by multi-headed CHR rules.

Definition 4 (Bounded Transaction Elimination). A bounded *transaction* is one that performs a finite, statically known number of derivation steps.

Let $\beta(P)$ be the elimination of bounded transactions from P obtained by applying the following steps to each bounded transaction in P .

1. Replace a bounded atomic transaction $\text{atomic}(G)$ by a new operation constraint C , where C has the same formal parameters as G .
2. Add a rule r to the program of the form $C \Leftarrow G$.
3. Unfold³ the rule r , until no more operation constraints appear in its body.

In summary, we can make the following general claim.

Theorem 6. The bounded transaction elimination $\beta(P)$ of an CHR^\star program P satisfies the Partial Erasure Correctness Criterion, i.e. $\forall C. \mathcal{S}^\star \llbracket P \rrbracket (C) = \mathcal{S}^\star \llbracket \beta(P) \rrbracket (C)$.

There are numerous examples satisfying this criterion. We refer to [13] for details.

³ We refer to [16] for a formal treatment of unfolding in CHR.

6.2 From CHR^{*} to CHR via Confluence Analysis

We only consider atomic transactions that are well-behaved, i.e. do not get stuck:

Definition 5 (Well-Behaved Constraints). *We say that C is well-behaved wrt. program P iff $S^{\star}[P](C)$ only contains data constraints, and no operation constraints or atomic transactions.*

The motivation for this is that, in general (e.g. for unbounded transactions), we cannot model stuck atomic transactions by dropping the `atomic()` wrapper.

An example of a stuck transaction is a bank transfer (Section 2.1) that attempts to overdraw an account. This transaction can be made into a well-behaved transaction, if we drop the guard in the withdraw rule and hence allow negative balances:

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

In the remainder of this paper, we assume that the programmer ensures well-behavedness, and focus on ensuring isolation for well-behaved constraints.

With the new semantics of withdrawal, the `atomic()` seems superfluous: any two consecutive transfers *commute*. Regardless of the order they are performed in, they yield the same final result. Hence, we can safely erase the wrapper which then leads to

$$\begin{aligned} & \text{transfer}(a1,a2, 100) \wedge \text{transfer}(a3,a4,150) \\ \wedge & \quad \text{transfer}(a5,a2, 200) \wedge \text{transfer}(a6,a1, 50) \end{aligned}$$

We can now concurrently execute `transfer(a3,a4,150)` and `transfer(a6,a1,50)` and then sequentially execute the remaining transfers.

The generalized notion of the above commutativity is *confluence* [1]. A CHR program is confluent if any derivation from the same initial goal yields the same final result, i.e. $\forall C \exists C'. S[P](C) = \{C'\}$.

Hence, we get the following result if an erased program is confluent:

Theorem 7 (Erasure for Confluent Transactions). *If the erasure $\epsilon(P)$ of an CHR^{*} program P is confluent, then the ECC is satisfied for all well-behaved constraints, i.e. $\forall C. S^{\star}[P](C) = S[\epsilon(P)](\epsilon(C))$, where C is well-behaved.*

6.3 Relaxing Confluence

Confluence is a very strong assumption and guarantees that isolation is not violated once `atomic()` wrappers are removed. In practice, confluence is not satisfied by many programs where the non-determinism is acceptable. In case (non)confluence can be explained as a serial execution of critical pairs among atomic operations, it is still safe to drop the `atomic()` wrapper.

Example 2. Recall the earlier shared-linked list example from Section 2.2. Most of the critical pairs among find and insert operations are observably joinable assuming that the data structure is a sorted linked list. The only non-joinable critical pair arises in case the insertion takes place on a “found” node: `nil(P) ∧`

$\text{insert}(X,P) \wedge \text{find}(X,P)$ and $\text{node}(Y,P,\text{Next}) \wedge \text{insert}(X,P) \wedge \text{find}(X,P)$ where $Y < X$. Depending on the order of execution we obtain different results. For example, for the first case the find fails if the insert is performed last. For the second case, it is the other way around. Yet each execution path corresponds to a valid serial execution of an atomic execution of find and an atomic execution of insert on a shared linked list. Hence, we argue that we can safely drop the `atomic()` wrapper around single find and insert operations.

We can restate Theorem 7 for partially confluent programs under two additional conditions.

Definition 6. A CHR program P is partially confluent iff all critical pairs are either observably joinable [4] wrt. an invariant⁴, or non-joinability can be explained as a serial execution of the operations involved.

Definition 7. A constraint C satisfies the single atomic operation property (SAOP) iff each `atomic()` wrapper contains at most one operation constraint and zero or more data constraints.

An $\text{CHR}^{\star\star}$ program P satisfies the single atomic operation property iff for each initial constraint satisfying SAOP, all constraints in intermediate derivations satisfy SAOP as well.

Theorem 8 (Erasure for Partially Confluent SAOP Transactions). If the erasure $\epsilon(P)$ of an $\text{CHR}^{\star\star}$ program P is partially confluent and P satisfies SAOP the ECC is satisfied for all well-behaved, SAOP constraints, i.e. $\forall C. \mathcal{S}^{\star\star}[P](C) = \mathcal{S}[\epsilon(P)](\epsilon(C))$, where C is well-behaved and satisfies SAOP.

The restriction to SAOP is essential as the following example shows.

Example 3. Execution of

$$\begin{aligned} &\text{node}(1,p,q) \wedge \text{nil}(q) \wedge \text{atomic}(\text{insert}(2,p) \wedge \text{insert}(4,p)) \\ &\quad \wedge \text{atomic}(\text{find}(2,p) \wedge \text{find}(4,p)) \end{aligned}$$

has two possible outcomes. We either find 2 and 4 or we find none of the two values. If we naively drop the `atomic()` wrappers it would however be possible to observe an intermediate of the first transaction where we find 2 but not 4.

We conclude that partial confluence is only a sufficient criteria to guarantee isolation for SAOP transactions (of course also in subsequent derivation steps). More complex transactions require a more involved confluence analysis.

6.4 Completion for Stuck Transactions

Confluence analysis is also useful to recover from stuck operations.

Example 4. Consider the extension of the linked list program with the operation `delete(X,P)`, which deletes X from the list (if present). This operation is implemented by the rules:

⁴ sortedness in the above case

```

d1 @ node(X,P,Q), node(Y,Q,Next), delete(X,P) <=> node(Y,P,Next).
d2 @ node(X,P,Q), nil(Q), delete(X,P) <=> nil(P).
d3 @ node(X,P,Q) \ delete(Y,P) <=> X < Y | delete(Y,Q).
d4 @ node(X,P,Q) \ delete(Y,P) <=> X > Y | true.
d5 @ nil(P) \ delete(X,P) <=> true.

```

Adding these rules, leads to additional critical pairs. Several are of the kind discussed in the previous section, e.g. a find before or after a delete yields a different result, but both are acceptable.

However, we get a new kind of critical pair, where an operation gets stuck. For instance, consider the goal:⁵

$$\text{node}(X,P,Q) \wedge \text{node}(Z,Q,R) \wedge \text{find}(Y,Q) \wedge \text{delete}(Y,Q)$$

where $X < Y < Z$. If the find makes step first and then the delete removes the node, we get $\text{node}(X,P,R) \wedge \text{find}(Z,R)$ and the find can continue looking at the nodes not shown. However, if the delete goes first, the find is stuck: $\text{node}(X,P,R) \wedge \text{find}(Z,Q)$. The problem is that there is no more node Q to look at, and so the find does not know how to proceed.

Such a critical pair is undesirable and has to be eliminated. We do so by a form of domain-specific *completion*, i.e. we add rules to the program that identify the stuck state and get out of it. Firstly, to facilitate recognizing (potentially) stuck states, the `delete` operation should leave a trace of its operation: the `delnode(Q,P)` data constraint denotes that there previously was a node at location P with predecessor at Q.

```

d1 @ node(X,P,Q), node(Y,Q,Next), delete(X,P) <=>
    node(Y,P,Next) ^ delNode(Q,P).
d2 @ node(X,P,Q), nil(Q), delete(X,P) <=>
    nil(P) ^ delNode(Q,P).

```

Now the stuck operation can be detected by matching against the appropriate `delnode` constraint, and the operation can continue properly.

```

f5 @ nodeDel(P,Q) \ find(X,P) <=> find(X,Q).
i5 @ nodeDel(P,Q) \ insert(X,P) <=> insert(X,Q).
d6 @ nodeDel(P,Q) \ delete(X,P) <=> delete(X,Q).

```

The above rules perform a *smart* retry. Instead of restarting (failing) the entire transaction from scratch, only a small number of steps are required to recover and continue the transaction. A similar form of domain-specific completion has been performed by Frühwirth for the “union transaction” of a parallel union-find algorithm [7]. In essence, Frühwirth’s starting point is a correct, parallel union-find implementation in CHR^{*}, although this is implicit in his work. He then performs sophisticated transformations to obtain a version in plain CHR.

⁵ For simplicity, we don’t show nodes beyond Q.

7 Related Work

The idea of using semantic analysis to achieve greater concurrency and better failure recovery for transactions can already be found in the database literature [17, 10]. Our contribution is to transfer these ideas to the programming language, specifically, Constraint Handling Rules setting.

Several other works enrich an existing programming language calculus with transactions. For example, the work in [2] considers a process calculus extended with atomic transactions whereas the work on transaction logic [3] adds transactional state updates to a Horn logic calculus. None of these works consider meaning-preserving transformation methods from the transactional calculus to the plain calculus. Yet many of the examples found in the above works belong to the class of bounded transactions and can therefore be transformed to plain multi-headed CHR rules.

Support for transactions is now generally available in many programming languages. The issue of “false” conflicts (see Section 5.1) is well-recognized but we are only aware of a few works [12, 8, 9] which address this important problem. They appear to be complementary to our work, i.e. the methods proposed could be integrated with our transformational approach. We discuss them in turn below.

One possible approach to avoid “false” conflicts is to release read logs early, e.g. see [12]. The problem with this method is that it requires great care by the programmer to preserve atomicity and isolation.

A more principled approach proposed in [8] is to re-run only those parts of a transaction which have been actually affected by “real” conflicts. This approach is well-suited for side-effect free, declarative languages such as CHR but still requires to keep track of read/write logs to infer the (un)affected parts.

Fairly close in spirit is the work in [9] which establishes formal criteria to boost the level of concurrency of transactional code. The starting point is a language with support for highly concurrent linearizable operations to which then transactions are added such that the transactional code exhibits the same level of concurrency as the underlying operations. In the context of our work, a highly concurrent linearizable operation can be viewed as a multi-headed CHR rule. If the operations commute, i.e. are confluent, they can be scheduled for concurrent execution. The main differences are that the work in [9] still requires a transaction manager to detect conflicts between non-commutative operations (and there is the potential of deadlocks as pointed out in [9]). Our goal is to avoid as much as possible the need for a transaction manager by performing the opposite transformation, from transactional to non-transactional code. In general, our approach cannot deal with non-commutative operations, we require the stronger condition of partial confluence. On the other hand, we can easily perform domain-specific optimizations by providing specialized synchronization and recovery rules (see Section 6.4).

8 Conclusion and Future Work

We have presented the CHR^{\star} calculus for CHR with atomic and isolated transactions. To execute CHR^{\star} efficiently, we propose a simple execution scheme, that

sequentializes atomic transactions. In order to improve concurrency, we propose several transformation methods that replace transactions by plain CHR goals. The methods are based on well-established concepts: unfolding and confluence analysis of CHR

In future work, we plan to investigate further transformations and to what extent they can be automated.

References

1. Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.
2. L. Acciai, M. Boreale, and S. Dal-Zilio. A concurrent calculus with atomic transactions. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 48–63. Springer, 2007.
3. A. J. Bonner and M. Kifer. Transaction logic programming. In *Proc. of ICLP'93*, pages 257–279, 1993.
4. G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable confluence for Constraint Handling Rules. In V. Dahl and I. Niemelä, editors, *ICLP '07*, volume 4670 of *LNCS*, pages 224–239. Springer, September 2007.
5. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
7. Thom Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence. In M. Gabbrielli and G. Gupta, editors, *ICLP '05*, volume 3668 of *LNCS*, pages 113–127, Sitges, Spain, October 2005. Springer.
8. T. Harris and S. Stipic. Abstract nested transactions, 2007. TRANSACT 2007: The Second ACM SIGPLAN Workshop on Transactional Computing.
9. M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. of PPOPP '08*, pages 207–216. ACM Press, 2008.
10. H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. Technical report, 1990.
11. Edmund S.L. Lam and Martin Sulzmann. Towards agent programming in CHR. In T. Schrijvers and Th. Frühwirth, editors, *CHR '06*, pages 17–31, July 2006.
12. Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of PPOPP '07*, pages 68–78. ACM Press, 2007.
13. T. Schrijvers and M. Sulzmann. Transactions in constraint handling rules, 2008. Technical Report.
14. Tom Schrijvers and Thom Frühwirth. Optimal union-find in Constraint Handling Rules. *TPLP*, 6(1–2):213–224, 2006.
15. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. Accepted at ACM TOPLAS, 2008.
16. P. Tacchella, M. Gabbrielli, and M. Chiara Meo. Unfolding in CHR. In M. Leuschel and A. Podelski, editors, *PPDP '07*, pages 179–186. ACM Press, July 2007.
17. W. E. Weihl. Data-dependent concurrency control and recovery. *SIGOPS Oper. Syst. Rev.*, 19(1):19–31, 1985.
18. G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

A Examples

A.1 Blocks World

We consider a set of CHR rules describing the behavior of agents (robot arms) in a blocks world [11].

```
get(R,X),empty(R),clear(X),on(X,Y) <=> holds(R,X),clear(Y)
putOn(R,X,Y),holds(R,X),clear(Y) <=> empty(R),on(X,Y),clear(X)
```

The meaning of the constraints should be obvious: `get/1` and `putOn/2` denote operations and the other predicates data. We make this distinction because we will be only interested in atomic execution of operations. The first rule gets an object `X` whereas the second rule puts an object `X` onto another object `Y`. The actual operations are performed as an atomic left-to-right rewriting step. For example, we can only get an object `X` if the robot arm `R` is clear, the object is not obstructed and stands on top of another object `Y`. The result is robot arm `R` holding object `X` and a cleared object `Y`. The above CHR guarantee the atomic and isolated execution of single operations but we cannot guarantee the same for multiple operations.

In the CHR^{*} calculus, we can on the other hand specify the atomic and isolated execution of multiple operations.

```
atomic(get(r1,block1),putOn(r1,block1,table1),
      get(r2,block2),putOn(r2,block2,table2))
```

It is clear that if there are only finite operations inside a `atomic()` wrapper, such programs belong to the class of bounded transactions and we can apply the unfolding technique discussed in Section 6.1 to remove the `atomic()` wrapper.

A.2 (Three) Dining Philosophers

```
think(X),fork(Y),fork(Z) <=> x = y /\ z = z+1 mod 3 | eat(X,Y,Z)
eat(X,Y,Z) <=> timeout(100) | think(X),fork(Y),fork(Z)
```

No need for `atomic()`, we only need multi-headed CHR rules/bounded transactions.

A.3 Union Find

```
make(E) <=> root(E).
```

```
union(A,B) <=> atomic(find(A,A),find(B,B),link(A,B)).
```

```
edge(X,Y) \ find(A,X) <=> find(A,Y).
root(X) \ find(A,X) <=> found(A,X).
```

```
link(A,B),found(A,R),found(B,R) <=> true.
```

```
link(A,B),found(A,RA),root(RA),
found(B,RB),root(RB) <=> root(RA),edge(RB,RA).
```

via confluence analysis, see [7], we obtain

```

make(E) <=> root(E).

union(A,B) <=> find(A,A), find(B,B), link(A,B).

edge(X,Y) \ find(A,X) <=> find(A,Y).
root(X)    \ find(A,X) <=> found(A,X).

edge(X,Y) \ found(A,X) <=> find(A,Y).

link(A,B), found(A,R), found(B,R) <=> true.

link(A,B), found(A,RA), root(RA),
found(B,RB), root(RB) <=> root(RA), edge(RB,RA).

```

B Well-Behavedness

Confluence is not sufficient to guarantee atomicity/well-behavedness. Below is a simple example.

```

sellOp(X), buyData(X) <=> true.

    atomic(sellOp(a) /\ buyData(b))

/\ atomic(buyData(c) /\ sellOp(b))

```

Above CHR rule is confluent. If we drop the `atomic()` wrappers, the first transaction will not be fully executed.

C Parallel Speed-ups

MARTIN

I'm wondering whether to postpone this section till later, when we actually transform CHR* to CHR, then the issue of maximizing parallelization becomes an issue.

Sneyers et al. [] use the notion of *derivation length*, i.e. the number of semantic steps of a derivation, as a measure for CHR's runtime. The aim of parallel execution of CHR is of course to improve the runtime over sequential execution. Hence, we define a derivation length for parallel CHR derivations for comparison with the sequential derivation length.

Unfortunately, the above semantics rules are rather unsuitable for precisely capturing the intuitive notion of parallel derivation length. Hence, we first propose the following alternate semantics, comprising a single rule:

$$\boxed{
 \begin{array}{c}
 \text{(GENPAR)} \quad \frac{E \wedge \bigwedge_{j \neq i} S_j \wedge S_i \rightsquigarrow E \wedge \bigwedge_{j \neq i} S_j \wedge S'_i}{E \wedge \bigwedge_{i=1}^n S_i \rightsquigarrow_{\parallel} E \wedge \bigwedge_{i=1}^n S'_i}
 \end{array}
 }$$

Now the number of \mapsto_{\parallel} steps faithfully reflects the runtime of a parallel CHR derivation.

Every sequential derivation is also a valid parallel derivation. Hence, the upperbound of a queries derivation length is not affected when going from the sequential to the parallel semantics. However, the lowerbound may be greatly affected.

Example 5. Consider the program below and a queries comprising of n `a` constraints.

```
a <=> true.
```

In the sequential semantics, the derivation length for this queries is n . In the parallel semantics, the upperbound on the derivation length is also n . However, the lowerbound is 1, the case where all n `a` constraints simultaneously simplify to `true`.

We will extend this notion of parallel derivation lengths to transactions below, and use it to reason over the maximum parallelism present in a program.