HAL
open science

Une sémantique complète pour CHR en logique des transactions

Marc Meister, Khalil Djelloul, Jacques Robin

To cite this version:

Marc Meister, Khalil Djelloul, Jacques Robin. Une sémantique complète pour CHR en logique des transactions. Troisièmes Journées Francophones de Programmationpar Contraintes (JFPC07), Jun 2007, INRIA, Domaine de Voluceau, Rocquencourt, Yvelines France. inria-00151149

HAL Id: inria-00151149

https://inria.hal.science/inria-00151149v1

Submitted on 1 Jun 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Une sémantique complète pour CHR en logique des transactions*

Marc Meister[1] **Khalil Djelloul**[1] **Jacques Robin**[2]

[1] Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm, Allemagne
[2] Centro de Informática, Universidade Federal de Pernambuco
Caixa Postal 7851, 50732-970 Recife PE, Brésil

Abstract

Reasoning on Constraint Handling Rules (CHR) programs and their executional behaviour is often ad-hoc and outside of a formal system. This is a pity, because CHR subsumes a wide range of important automated reasoning services. Mapping CHR to Transaction Logic ($\mathcal{TR}$) combines CHR rule specification, CHR rule application, and reasoning on CHR programs and CHR derivations inside one formal system which is executable. This new $\mathcal{TR}$ semantics obviates the need for disjoint declarative and operational semantics.

1 Introduction

Constraint Handling Rules (CHR) [6] is a concurrent, committed-choice, rule-based language which was originally created as a declarative logic constraint language to implement monotonic reasoning services. Its main features are guarded rules which transform multi-sets of constraints (atomic formulas) into simpler ones until they are solved.

Over the last decade, CHR has become available for most Prolog systems, Java, Haskell, and Curry and has matured into a general-purpose programming language with many applications [12] : Nonmonotonic reasoning services can be implemented in CHR, e.g. the fluent executor (FLUX) [13] which provides general reasoning facilities about actions and sensor information under incomplete information. Also, classic algorithms like the union-find, which rely on inherently nonmonotonic updates, have been implemented with optimal complexity in CHR [11].

The *operational semantics* of CHR is specified by a state transition system. Although applicability of a CHR rule is defined within predicate logic, the operational semantics is not integrated into a logic and is different from the declarative semantics in predicate logic. Basically the problem is that there is no elegant *predicate logic-based semantics* for changing the constraint store. Hence, reasoning on CHR programs and their executional behaviour is often ad-hoc and outside of a formal logic-based system.

We integrate the operational semantics of CHR into Transaction Logic ($\mathcal{TR}$) [3, 4, 5] which extends predicate logic with – among other things – a declarative account for state changes in logic programs (cf. [3] for a list of failed attempts to formalise updates in a logic programming language). Transaction Logic naturally enjoys nonmonotonic behaviour due to the dynamics of a database which represents a current state [8].

Contributions and overview of the paper. By mapping the core of CHR to $\mathcal{TR}$, we combine CHR rule specification, CHR rule application, and reasoning on CHR programs and CHR derivations inside one formal system which is executable. We show that a CHR rule applies if and only if the $\mathcal{TR}$ query of the mapping of this CHR rule succeeds in $\mathcal{TR}$ and extend this result to CHR derivations by integrating the CHR run-time system. A formal statement then links the procedural aspect of execution (the operational semantics) with a new model-theoretic (declarative) reading. Thus our semantics covers both operational and declarative aspects elegantly. An efficient proof system in $\mathcal{TR}$ exe-

*Paper is accepted at Ninth International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2007.

cutes CHR programs and reasons on CHR derivations mechanically.

- We present the aspect of a missing unified semantics for CHR through an easy example in Section 2 and propose our solution to overcome this missing aspect in Section 3.
- We explain the most basic instantiation of $\mathcal{TR}$ to give a logical account for range-restricted ground CHR programs in Section 4.
- We map the constraint store to a database, the CHR program to a serial-Horn $\mathcal{TR}$ program that updates this database, and the CHR run-time system to a generalised-Horn $\mathcal{TR}$ program. The details of our CHR-to-$\mathcal{TR}$ mapping in Section 5 are necessary for our sound- and completeness result which is our main contribution.
- In Section 6 we apply our approach to two examples, showing how to execute and reason on them in the framework of Transaction Logic. We use the FLORA-2 system [14] for implementation.

Complete proofs and full CHR and FLORA-2 sources of the examples are available at `http://www.informatik.uni-ulm.de/pm/index.php?id=138`.

2 The Problem : Reasoning on Constraint Handling Rules

Example 1 Consider a coin-throw simulation program[1], consisting of two CHR rules r_1 and r_2.

$$r_1 \ @ \ \texttt{throw} \Leftrightarrow \texttt{caput} \qquad r_2 \ @ \ \texttt{throw} \Leftrightarrow \texttt{nautica}$$

Intuitively, as both rules are applicable for the goal `throw`, the answer constraint is `caput` or `nautica` depending on the rule selection. Clearly, we have the two possible state transitions $(\texttt{throw}) \rightarrowtail_{r_1} (\texttt{caput})$ and $(\texttt{throw}) \rightarrowtail_{r_2} (\texttt{nautica})$ for the goal `throw`.

What we are missing is one logic-based formal system for mechanical *execution* and *reasoning*, which should be implemented to also allow automatic reasoning. Available CHR run-time systems (e.g. the reference implementation in SICStus Prolog for CHR) come as black-boxes and offer no means for reasoning. For example, we want to prove the following three properties automatically :

(P1) Throwing a coin can yield `caput`.

(P2) Throwing a coin cannot yield both `caput` and `nautica`.

(P3) Application of r_1 cannot yield `nautica`.

[1] To avoid misunderstandings with the *head* of a rule, we replaced the good old "head" and "tail" with the ancient "caput" and "nautica".

Because the constraint `throw` is interpreted as a trigger (and not as static truth) in the coin-throw simulation program, the gap between the predicate logic declarative semantics [6] of this general-purpose CHR program – the meaningless formula $\texttt{caput} \leftrightarrow \texttt{nautica}$ – and its executions is especially large. The underlying problem is that predicate logic is a static logic, unable to express the dynamics of deletion and insertion directly. Here, reasoning has to be done ad-hoc (outside of a logic) along the operational semantics of CHR [1].

The linear logic semantics [2] overcomes this restriction of the classic predicate logic semantics and gives a meaningful declarative semantics also for general-purpose CHR programs. While the linear logic notion of a resource models the necessary dynamics, it does not cover all aspects of the operational semantics : Linear logic has no inherent notion of execution and we cannot reason on the execution itself but only on the result of an execution. Similar to the classic declarative semantics, the linear logic semantics links initial and final state with a logical reading of the program. As CHR derivations are mimicked inside its proofs, reasoning on derivations is not possible directly.

Summarising, both predicate and linear logic *declarative* semantics allow reasoning on the properties of the program, but lack the possibility to actually execute the rules, reason on the execution, and are not readily mechanised. Thus, reasoning on execution lacks a formal logic-based framework. Most importantly, specification (as CHR rules), execution (by a CHR run-time system), and reasoning are not integrated and reasoning is either done by hand or by special-purpose tools (e.g. for *confluence* [1]). The need to integrate the operational semantics into a logic was recognised by Maher [9] : Besides a "logical" (declarative) semantics, also a data/control-flow analysis is highly desirable, e.g. to prove termination of a program. Clearly this data/control-flow analysis aspect is inherently absent in [6, 2] which cover the "logical" (declarative) semantics only. Maher continues, that "there is possibility that this analysis can be carried out within a logic framework"[9, p. 870]. We argue that Transaction Logic ($\mathcal{TR}$) provides this missing aspect in the next section.

3 The Idea : Map CHR to Transaction Logic

We map CHR to Transaction Logic to simulate the operational semantics of CHR by logic programming with state changes and use executional entailment – a formal statement in $\mathcal{TR}$ – to execute and to reason on CHR derivations. In their seminal work on Transaction Logic [3], Bonner and Kifer extend predicate

logic with a declarative account for state changes in logic programming. As the operational semantics of CHR is formalised by a state transition system, where a CHR rule application changes the constraints store, we map CHR programs to serial-Horn $\mathcal{TR}$ programs and identify the application of a CHR rule by the state transition system with a successful query of the $\mathcal{TR}$ program. To this end, we map the constraint store to a database with the elementary database updates *insertion* and *deletion*. A CHR derivation is then the side-effect on the database when the $\mathcal{TR}$ proof system infers the $\mathcal{TR}$ query to be true.

Example 1 (Cont.) We show the basic ideas for the coin-throw simulation program with a non-deterministic rule selection strategy (and review this example in Section 6 in detail). We map rule r_1 to the serial-Horn $\mathcal{TR}$ rule $r_1^{\mathcal{TR}}$.

$$r_1^{\mathcal{TR}} \equiv \mathrm{chr}(r_1) \leftarrow \texttt{throw} \otimes \texttt{throw}.\mathit{del} \otimes \texttt{caput}.\mathit{ins}$$

To make the $\mathcal{TR}$-predicate $\mathrm{chr}(r_1)$ true, we have to execute the serial conjunction on its right hand side : First check that `throw` is present, then delete it, and then insert `caput`. The order in the serial conjunction $\otimes$ is crucial, as the $\mathcal{TR}$-predicates $\texttt{throw}.\mathit{del}$ and $\texttt{caput}.\mathit{ins}$ have side-effects on the database[2]. If we execute $\mathrm{chr}(r_1)$ on the (initial) database $\{\texttt{throw}\}$, we pass through the empty database $\{\}$, and arrive at the (final) database $\{\texttt{caput}\}$. For $P = \{r_1^{\mathcal{TR}}\}$, we have the following *executional entailment* statement $\models_x$ in $\mathcal{TR}$, which states, that the successful invocation of program P by $\mathrm{chr}(r_1)$ can successfully update the database along the given *execution path* $\{\texttt{throw}\}, \{\}, \{\texttt{caput}\}$.

$$P, \{\texttt{throw}\}, \{\}, \{\texttt{caput}\} \models_x \mathrm{chr}(r_1)$$

The executional entailment statement has both a procedural (operational) and a model-theoretic (declarative) semantics in $\mathcal{TR}$. On the one hand, an available efficient $\mathcal{TR}$ inference system for the subclass of serial-Horn programs actually computes the necessary updates of an initial database $\{\texttt{throw}\}$ when establishing the truth of $\mathrm{chr}(r_1)$ and implements the procedural aspect of $\mathcal{TR}$. Integrating the operational semantics of CHR into $\mathcal{TR}$ by executional entailment, we have – on the other hand – a new model-theoretic (declarative) semantics which captures the possible executions of a CHR program.

We show in Section 5, that a CHR rule r is applicable iff we can establish the truth of the head of a $\mathcal{TR}$ rule $r^{\mathcal{TR}}$ and then extend our mapping to cover the CHR run-time system. The changes caused on the constraint store are mapped one-to-one to updates of the database as we simulate CHR rule application by the $\mathcal{TR}$ inference system.

We can then prove properties **(P1-3)** from Section 2 mechanically. Even better, the FLORA-2 system allows us to both execute and reason on this example automatically (cf. Section 6).

4 Preliminaries

We provide necessary background for readers not familiar with CHR and $\mathcal{TR}$.

4.1 Constraint Handling Rules

Constraint Handling Rules (CHR) [6, 12] is a concurrent, committed-choice, rule-based logic programming language. We distinguish between two different kinds of constraints : *built-in constraints* which are solved by a given constraint solver, and *user-defined (CHR) constraints* which are defined by the rules in a CHR program. This distinction allows one to embed and utilise existing constraint solvers as well as side-effect-free host language statements. As we trust the built-in black-box constraint solvers, there is no need to modify or inspect them.

A *CHR program* is a finite set of rules. There are two main kinds of rules : *Simplification rules* $R \mathbin{@} H \Leftrightarrow G \mid B$ and *propagation rules* $R \mathbin{@} H \Rightarrow G \mid B$. Each rule has a unique name R, the *head* H is a non-empty multi-set conjunction of CHR constraints, the *guard* G is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a multi-set conjunction of built-in and CHR constraints. A trivial guard expression "*true* |" can be omitted.

Since we do not focus on propagation rules in this paper, it suffices to say that they are equivalent (in the standard semantics) to simplification rules of the form $R \mathbin{@} H \Leftrightarrow G \mid (H \wedge B)$.

The *operational semantics* of CHR is defined by a state transition system where states are conjunctions of constraints. To a conjunction of constraints, rules are applied until a fixpoint is reached. Note that conjunctions in CHR are considered as *multi-sets* of atomic constraints. Any of the rules that are applicable can be applied and rule application cannot be undone since CHR is a committed-choice language. A simplification rule $R \mathbin{@} H \Leftrightarrow G \mid B$ is applicable in state $(H' \wedge C)$, if the built-in constraints C_b of C imply that H' matches the head H and the guard G is entailed under this matching, cf. (1). The consistent, predicate logic, built-in constraint theory CT contains

[2]The symbol $\otimes$ stands for serial conjunction in $\mathcal{TR}$ and *not* for join of views on databases.

Clark's syntactic equality.

$$\begin{array}{ll} \text{IF} & R \,@\, H \Leftrightarrow G \mid B \text{ is a fresh variant of} \\ & \text{rule } R \text{ with variables } \bar{X} \\ \text{AND} & CT \models (\forall)\, C_b \rightarrow \exists \bar{X}\, (H = H' \wedge G) \\ \text{THEN} & (H' \wedge C) \rightarrowtail_R (B \wedge G \wedge H = H' \wedge C) \end{array} \quad (1)$$

If applied, a simplification rule *replaces* the matched CHR constraints in the state by the body of the rule. In the operational semantics, rules are applied until exhaustion, i.e. the CHR run-time system (which actually runs a CHR program by selecting applicable rules and matching constraints) computes the reflexive transitive closure $\rightarrowtail^*$ of $\rightarrowtail$. The CHR run-time system should stop immediately, when insertion of a built-in constraint makes C_b inconsistent. However, this *termination at failure* is not explicitly addressed in the operational semantics.

4.2 Transaction Logic

Transaction Logic ($\mathcal{TR}$) [3, 4, 5] is a conservative extension of classical predicate logic, where predicates can have side-effects on a database, allowing to model state changes. Similar to predicate logic, $\mathcal{TR}$ features a Horn fragment which supports logic programming. While $\mathcal{TR}$ is an extremely versatile logic to handle specification, execution, and reasoning on logic programs with updates, it suffices for this work to use a basic instantiation of $\mathcal{TR}$ which restricts side-effects to the updates *insertion* and *deletion* on a relational, ground database.

A database is a set of ground atoms. A sequence of databases $D_0, \ldots, D_n$ is called a *path* $\pi = \langle D_0, \ldots, D_n \rangle$ which can be *split* into sub-paths $\langle D_0, \ldots, D_i \rangle \circ \langle D_i, \ldots, D_n \rangle$ (for $0 \leq i \leq n$). Access to the database is restricted by two oracles : The *data oracle* $\mathcal{O}^d$ maps a database D to a set of ground atoms that are considered to be true along the path $\langle D \rangle$. Elementary database updates are captured by the *transition oracle* $\mathcal{O}^t$ which maps two databases D and D' to a set of ground atoms considered to be true along the path $\langle D, D' \rangle$.

Definition 1 (Path Structure with Relational Oracles) *A* path structure M *assigns a classical Herbrand structure (or* $\top$ *which satisfies everything) to every path and is subject to the following restrictions for ground atoms* p.

$$\begin{array}{ll} M(\langle D \rangle) \models p & \text{if } p \in D \\ M(\langle D, D' \rangle) \models p.ins & \text{if } D' = D \cup \{p\} \\ M(\langle D, D' \rangle) \models p.del & \text{if } D' = D \setminus \{p\} \end{array} \quad (2)$$

The relational oracles are given by $p \in \mathcal{O}^d(D)$ *iff* $p \in D$, $p.ins \in \mathcal{O}^t(D, D')$ *iff* $D' = D \cup \{p\}$, *and* $p.del \in \mathcal{O}^t(D, D')$ *iff* $D' = D \setminus \{p\}$.

Quantification of $\mathcal{TR}$ formulas and satisfaction of composed $\mathcal{TR}$ formulas are defined analogously to predicate logic : A $\mathcal{TR}$ formula with $\neg, \wedge, \vee$, or $\leftarrow$ as main connective is *satisfied along a path* π if the appropriate property holds between its sub-formulas *along the same path* π. Satisfaction from the basic properties (2) extends to the case of longer paths by the new *serial conjunction* operator : A serial conjunction $\phi \otimes \psi$ is satisfied along the path π iff ϕ is satisfied along π_1 and ψ is satisfied along π_2 for *some* split of the path $\pi = \pi_1 \circ \pi_2$. The modal *possibility* $\Diamond\phi$ expresses that ϕ is satisfiable along *some* path starting from the current database, formally $M(\langle D \rangle) \models \Diamond\phi$ iff there is a path π starting at database D with $M(\pi) \models \phi$.

The formal statement *executional entailment* links a program, a possible sequence of databases which captures the side-effects of the program, and the invocation of the program.

Definition 2 (Executional Entailment) *Consider a set of* $\mathcal{TR}$ *formulas* P *and an* execution path *consisting of a sequence of databases* $D_0, \ldots, D_n$. *A path structure* M_P *is a* model of P *iff* $M_P(\pi) \models \phi$ *for every* $\mathcal{TR}$ *formula* $\phi \in P$ *and every path* π.

$$\begin{array}{l} P, D_0, \ldots, D_n \models_x \phi \;\; \textit{iff} \\ M_P(\langle D_0, \ldots, D_n \rangle) \models \phi \textit{ for every model } M_P \textit{ of } P \end{array}$$

Executional entailment, $\models_x$, selects one of (possibly several) valid execution paths, for which ϕ is true for all models M_P of P. A model M_P of P is a path structure that respects the oracles and satisfies every formula of P along every path. The execution path $D_0, \ldots, D_n$ records all side-effects when establishing the truth of ϕ, i.e. the "successful program invocation of ϕ for program P *can* update the database along execution path from D_0 to $D_1 \ldots$ to D_n" [3, p. 31].

Example 2 Consider the $\mathcal{TR}$ program $P = \{r \leftarrow p \otimes p.del\}$. Invocation of r deletes p from the database, but only if p was initially present and we have $P, \{p, q\}, \{q\} \models_x r$ but $P, \{q\}, \{\} \not\models_x r$. Also, deletion of p should be the only side-effect of the invocation of r, hence $P, \{p, q\}, \{\} \not\models_x r$ as $M_P(\langle \{p, q\}, \{\} \rangle) \models r$ is not correct for *all* models M_P of P. When we insert q under the condition that r *can* execute, e.g. $P, \{p\}, \{p, q\} \models_x \Diamond r \otimes q.ins$, we keep p in the database.

For the class of *serial-Horn programs* (i.e. sets of Horn rules with only serial conjunctions in the r.h.s) and *serial queries*, $\mathcal{TR}$ features an *executional deduction inference system* [3]. For a serial-Horn program P, an initial database D_0, and an existentially quantified

serial query $(\exists)\,\phi$, it infers the *sequent* P, D_0--- $\vdash$ $(\exists)\,\phi$ iff there is an executional entailment of $(\exists)\,\phi$ along an *execution path* starting from D_0. Most importantly the system both tries to infer the truth of $(\exists)\,\phi$ and computes the necessary changes to D_0 which we record in $D_0, \ldots, D_n$. Formally, the following fundamental sound- and completeness result links the model-theoretic executional entailment with the mechanised executional deduction.

Theorem 1 (Bonner and Kifer [3])
$P, D_0, \ldots, D_n \models_x (\exists)\phi$ iff there is an executional deduction of $(\exists)\phi$ with execution path $D_0, \ldots, D_n$.

For our serial-Horn program P from Example 2 we have $P, \{p, q\}$--- $\vdash$ r and the successful inference of the query r computes the execution path $\{p, q\}, \{q\}$ from the initial database $\{p, q\}$. Of course, we cannot infer $P, \{q\}$--- $\vdash$ r as there exists no execution path for the query r starting from $\{q\}$. By the definition of executional entailment, an execution either succeeds or all tentative side-effects are rolled back. Due to this *transaction property* of $\mathcal{TR}$ we cannot infer $P, \{p\}$--- $\vdash$ $r \otimes r$ and the tentative deletion of p by the first call to r is not manifested as the second call to r fails.

5 The Details : CHR-to-$\mathcal{TR}$-Mapping

We map CHR states to databases, adapt the data oracle $\mathcal{O}^d$, map CHR rules to serial-Horn $\mathcal{TR}$ rules, and specify the CHR run-time system as a generalised-Horn $\mathcal{TR}$ program. We then show our sound- and completeness result that links CHR derivations with executional entailment statements of $\mathcal{TR}$.

For this paper, we restrict ourselves to *range-restricted ground CHR*. Range-restricted CHR rules have no local variables, i.e. every variable in each rule already occurs in the head of the rule and all CHR states are ground as there are no variables in the goal.

5.1 Mapping CHR States to Valid Databases

We map each ground, user-defined constraint c_i of a CHR state (recall that a CHR state is a multi-set conjunction) to a $\mathcal{TR}$-predicate $u(c_i, i)$ where the second argument is a unique identifier – we use a natural number. We trail a new, unique identifier k in a bookkeeping $\mathcal{TR}$-predicate $n(k)$ (assuming that $u/2$ and $n/1$ are not defined by CT). Reflecting user-defined constraints as $\mathcal{TR}$-function symbols allows us to specify the necessary bookkeeping for the insertion and deletion of user-defined constraints as a serial-Horn program.

Definition 3 *A* valid database *D contains one bookkeeping predicate $n(k)$, predicates $u/2$ with unique identifiers that are all smaller than k, and built-ins b_i. The mapping $m_{\mathtt{s}}$ is defined from the set of CHR states $\mathcal{S}$ (consisting of user-defined constraints c_i and built-ins b_i) to the set of valid databases $\mathcal{D}$ by*

$$(\bigwedge_{0 \le i < k} c_i \wedge \bigwedge_{0 \le i < l} b_i) \mapsto$$
$$\{u(c_i, i) : 0 \le i < k\} \cup \{n(k)\} \cup \{b_i : 0 \le i < l\} \ .$$

Two valid databases are equivalent*, $\sim$, iff they differ only in the set of identifiers (including the argument of the bookkeeping predicate) and there is a bijective mapping between these sets.*

Clearly, $m_{\mathtt{s}}(c(a)) = \{u(c(a), 0), n(1)\}$ and $\{u(c(a), 5), n(9)\}$ are equivalent. We update valid databases through the serial-Horn program $P_{\mathtt{basic}}$:

$$\begin{aligned} \text{udel}(U) &\leftarrow u(U, K) \otimes u(U, K).del \\ \text{uins}(U) &\leftarrow n(K) \otimes n(K).del \otimes n(K+1).ins \otimes \\ &\qquad u(U, K).ins \end{aligned} \tag{3}$$

Deletion of a ground user-defined constraint is conditional (cf. Example 2) and insertion requires some bookkeeping.

Property 1 (Conditional Deletion of User-Defined Constraints) Invocation of udel(c) deletes a copy of the ground, user-defined constraint c from the valid database $D + \{u(c, k)\}$[3] and terminates in the valid database D.

$$P_{\mathtt{basic}}, D + \{u(c, k)\}, D \models_x (\exists)\ \text{udel}(c) \text{ with } k \in \mathbf{N}$$

Property 2 (Insertion of User-Defined Constraints) Invocation of uins(c) inserts a new copy of the ground, user-defined constraint c into the valid database $D + \{n(k)\}$ and terminates in the valid database $D + \{n(k+1), u(c, k)\}$.

$$P_{\mathtt{basic}}, D + \{n(k)\}, \ldots, D + \{n(k+1), u(c, k)\} \models_x (\exists)\ \text{uins}(c) \text{ with } k \in \mathbf{N}$$

5.2 Mapping the Built-In Theory CT to the Data Oracle $\mathcal{O}^d$

For range-restricted ground CHR the entailment condition of the state transition system, defined in (1) can be simplified as there are no local variables. Because we match the head H with the ground

[3] We use "+" to denote disjoint set union.

constraints H' from the store, the formula $\exists \bar{X}(H = H' \wedge G)$ is ground. We extend the relational data oracle $\mathcal{O}^d$, defined in (2), to implement the built-in constraint theory CT.

Definition 4 (Data Oracle as Built-in Solver) *For any database D and ground atomic built-in ϕ, the data oracle respects CT :*
$\phi \in \mathcal{O}^d(D)$ if $CT \models D_b \rightarrow \phi$ for the conjunction D_b of built-in predicates of D.

5.3 Mapping CHR Rules to Serial-Horn Rules in $\mathcal{TR}$

We map CHR rules to $\mathcal{TR}$ rules that update the database through $P_{\texttt{basic}}$ as defined in (3). We normalise any range-restricted CHR rule to contain no function symbols in the head by introducing a new variable for any implicit equality in the head and adding an explicit (new) equality to the guard, e.g. $r \mathbin{@} p(a) \Leftrightarrow true$ normalises to $r \mathbin{@} p(X) \Leftrightarrow X = a \mid true$.

Definition 5 *Consider a normalised range-restricted simplification rule r. The head H of rule r is the (multiset) conjunction $\bigwedge_{i=0}^{n_h} h_i$ of user-defined constraints h_i, the guard G is the conjunction $\bigwedge_{i=1}^{n_g} g_i$ of built-in constraints g_i ($n_g = 0$ represents true), and the body B is the (multiset) conjunction $\bigwedge_{i=1}^{n_u} u_i$ of constraints u_i ($n_u = 0$ represents true). The auxiliary t maps user-defined constraints u_i to $\mathrm{uins}(u_i)$ and built-in constraints to $u_i.ins$. We define the mapping $m_{\mathtt{r}} : r \mathbin{@} H \Leftrightarrow G \mid B \mapsto r^{\mathcal{TR}}$ by*

$$r^{\mathcal{TR}} \equiv \mathrm{chr}(r) \leftarrow \left(\bigotimes_{i=0}^{n_h} \mathrm{udel}(h_i)\right) \otimes \left(\bigotimes_{i=1}^{n_g} g_i\right) \otimes \left(\bigotimes_{i=1}^{n_u} t(u_i)\right) \quad . \tag{4}$$

Our mapping $m_{\mathtt{r}}$ is guided by the intuition that establishing the truth of $\mathrm{chr}(r)$ should have the same effect on the database as rule application by CHR's state transition system on the constraint store. The body of $\mathrm{chr}(r)$ consists of parts corresponding to head, guard, and body of the CHR rule r : First, we succinctly query the database for copies of each head constraint and delete them. Then we pass the check for the guard (as r is range-restricted, all variables in the guard are now bound) to our data oracle which respects the built-in constraint theory CT. Finally, we add the body constraints, labelling each inserted user-defined constraint with a new identifier.

By the transaction property of $\mathcal{TR}$ we can safely intertwine applicability checks with updates of the database, e.g. if the guard fails, the tentative deletions of the user-defined head constraints are undone.

Formally, application of r by the state transition system is equivalent to executional entailment of $\mathrm{chr}(r)$ modulo identifier renaming.

Lemma 1 *Consider two ground CHR states S and S', two valid databases D and D', a normalised range-restricted CHR simplification rule r, and its mapping $r^{\mathcal{TR}}$ as defined in (4). For $D \sim m_{\mathtt{s}}(S)$ and $D' \sim m_{\mathtt{s}}(S')$ we have*

$$S \rightarrowtail_r S' \;\; \textit{iff} \;\; P_{\texttt{basic}} + \{r^{\mathcal{TR}}\}, D, \ldots, D' \models_x (\exists)\ \mathrm{chr}(r) \quad .$$

5.4 Sound and Complete : CHR Run-Time System in $\mathcal{TR}$

We now extend Lemma 1 from a single rule step of a single CHR rule to a CHR derivation of a CHR program by integrating the fixpoint computation, i.e. the operational semantics of CHR, into $\mathcal{TR}$.

Our main result shows that our mapping from CHR to $\mathcal{TR}$ is sound- and complete w.r.t. the operational semantics of CHR. To this end, we express *applicability of a CHR rule* in state S by $P, D \models_x (\exists) \Diamond \mathrm{chr}(R)$ with $D \sim m_{\mathtt{s}}(S)$ and use induction on the derivation length for the extension from $\rightarrowtail$ to $\rightarrowtail^n$.

Theorem 2 (Sound- and Completeness) *Consider two ground CHR states S and S', two valid databases D and D', a CHR program P consisting of range-restricted simplification rules, and its mapping $P^{\mathcal{TR}} = P_{\texttt{basic}} + \{r^{\mathcal{TR}} : r \in P\}$. For $D \sim m_{\mathtt{s}}(S)$, $D' \sim m_{\mathtt{s}}(S')$, and an execution path π starting in D and ending in D' we have*

$$S \rightarrowtail_P^* S' \;\; \textit{iff} \;\; P^{\mathcal{TR}}, \pi \models_x (\exists) \left(\bigotimes_{i=1}^{n} \mathrm{chr}(R_i)\right) \otimes [\neg\Diamond\, \mathrm{chr}(R)] \;\; \textit{with } n \in \mathbf{N}$$

where $[\neg\Diamond\, \mathrm{chr}(R)]$ restricts satisfaction of $\neg\Diamond\, \mathrm{chr}(R)$ to paths of length one.

We now sketch how to implement the CHR runtime system as $\mathcal{TR}$ program with hypothetic goals (to express possibility) and negated goals (to check that no rule is applicable). We capture the fixpoint semantics of CHR as

$$\mathrm{fixpoint} \leftarrow \textit{while}\ \mathrm{applicable}\ \textit{do}\ \mathrm{chr}(R)\ \textit{od}$$

and implement the imperative *while*-loop programming construct as a simple *generalised-Horn program*

$P_{\texttt{runTime}}$ in $\mathcal{TR}$.[4]

$$\text{fixpoint} \leftarrow \text{chr}(R) \otimes \text{fixpoint} \quad (5)$$

$$\text{applicable} \leftarrow \Diamond\, \text{chr}(R) \quad (6)$$

$$\text{fixpoint} \leftarrow [\neg\, \text{applicable}] \quad (7)$$

Rule (5) succeeds if the call chr(R) – which successfully applies a CHR rule R – succeeds. In this case we call fixpoint (tail-recursively). We need two generalised-Horn rules to express that no CHR rule is applicable : Rule (6) succeeds if a CHR rule is applicable and this test leaves the database D untouched and rule (7) succeeds if no CHR rule is applicable at the current state using *negation-as-failure* to compute [¬ applicable].

Bonner and Kifer give an extended (sound- and complete) executional deduction inference system that integrates the ◊ operator. Negation ¬ is then treated (outside of the proof system) as negation-as-failure. A slight modification of the model-theoretic executional statement allows to give a declarative account for locally stratified generalised-Horn programs. Compared to Definition 2, we no longer look at *all* but only at the *perfect models* of the program. As $P_{\texttt{runTime}}$ is stratified we can use this executional entailment statement, $\models_x^{\text{perf}}$, cf. [3].

Corollary 1 *Under the premises of Theorem 2 we have*

$$S \rightarrowtail_P^* S' \;\; \textit{iff} \;\; P^{\mathcal{TR}} + P_{\texttt{runTime}}, \pi \models_x^{\text{perf}} (\exists)\ \text{fixpoint} \quad .$$

Our *declarative $\mathcal{TR}$ semantics* of the CHR program P is the *perfect-model semantics of the generalised-Horn $\mathcal{TR}$ program* $P^{\mathcal{TR}} + P_{\texttt{runTime}}$. Invocation of fixpoint – on the other hand – computes $\rightarrowtail^*$ as side-effect on the database, i.e. captures the *operational semantics of CHR*. This brings together the operational and declarative semantics of CHR in $\mathcal{TR}$.

6 Examples

We use the FLORA-2 system [14], a sophisticated object-oriented, knowledge management environment that implements the executional deduction inference system of $\mathcal{TR}$ by offering backtrackable deletion and insertion of facts, to execute and reason on CHR. Similar to Prolog, but with handling updates in a declarative way, serial queries are treated from left to right and one database is kept at any time. We have $P, D_0, \text{---} \vdash (\exists)\, \phi$ iff the query "?- ϕ" succeeds for the program P from the initial database D_0. In this case, FLORA-2 updates the database according to the computed executional path as side-effect.

Example 3 (Coin-Throw Simulation Program) We revisit Example 1 in detail. For the program $P^{\mathcal{TR}}_{\texttt{coin}} = P_{\texttt{basic}} + \{r_1^{\mathcal{TR}}, r_2^{\mathcal{TR}}\}$, defined in (3) and by (4), and the initial database $m_{\texttt{s}}(\texttt{throw}) = \{u(\texttt{throw}, 0), n(1)\}$ we throw a coin by querying "?- chr(R)". This query succeeds, returns an answer substitution for R, and updates the database. In a subsequent query "?- $u(S, I)$" we query the current database state for the side S of the coin.

We now prove properties **(P1-3)** from Section 2 automatically :

(P1) The query "?- chr(R), $u(\texttt{nautica}, I)$"[5] succeeds from D_0, i.e. we have a mechanical proof that a computation (throw) $\rightarrowtail$ (nautica) exists. Due to the post-condition $u(\texttt{nautica}, I)$ the FLORA-2 system backtracks over rule-application if rule r_1 is selected in the first try.

(P2) Throwing a coin cannot yield both caput and nautica is true because the query "?- chr(R), $u(\texttt{nautica}, I), u(\texttt{caput}, J)$" fails (from database D_0).

(P3) Applying rule r_1 cannot yield nautica as "?- chr(r_1), $u(\texttt{nautica}, I)$" fails (from database D_0).

For complex CHR programs this knowledge is much less trivial and very valuable for understanding. While CHR programs are usually very concise, debugging is often tedious, and automatised reasoning is highly desirable.

Example 4 (Greatest Common Denominator) Euclid's algorithm to compute the greatest common denominator (gcd) is probably the first algorithm in history that is still commonly used. The CHR implementation of the gcd consists of only two rules, where the built-in theory CT also contains the order between natural numbers.

$$r_1 \;@\; \texttt{gcd}(0) \Leftrightarrow \textit{true}$$
$$r_2 \;@\; \texttt{gcd}(X_1) \wedge \texttt{gcd}(X_2) \Leftrightarrow 0 < X_1 \wedge X_1 \leq X_2 \mid \texttt{gcd}(X_1) \wedge \texttt{gcd}(X_2 \% X_1)$$

The CHR derivation $(\texttt{gcd}(24) \wedge \texttt{gcd}(30) \wedge \texttt{gcd}(42)) \rightarrowtail^* (\texttt{gcd}(6))$ computes the gcd of 24, 30, and 42. The gcd algorithm can be seen as a (very basic) nonmonotonic reasoning service, as e.g. adding gcd(7) to the goal invalidates the original answer constraint gcd(6).

[4] Note that we can add *termination at failure* by adding "D_b consistent" to the loop condition easily, allowing to reason also on *failed derivations*.

[5] The serial conjunction operator ⊗ is written as comma in FLORA-2.

As FLORA-2 does not implement the possibility operator $\Diamond$, we carry out the next two inferences mechanically – but not automatically. We now assume $D_0 = m_{\mathtt{s}}(\mathtt{gcd}(24) \wedge \mathtt{gcd}(30) \wedge \mathtt{gcd}(42))$ as initial database and simulate one (of several possible) CHR derivations – recall that CHR is a committed-choice language – by inferring the sequent $P^{\mathcal{TR}}_{\mathtt{gcd}} + P_{\mathtt{runTime}}, D_0$--- $\vdash$ $(\exists)$ fixpoint. Then we inspect the final database D_n of the computed executional path $D_0, \ldots, D_n$ which contains $u(\mathtt{gcd}(6), k)$ with some identifier $k \in \mathbf{N}$. Similarly, we have a mechanical proof that no $\mathtt{gcd}(0)$ constraint is in the final constraint store as we cannot infer $P^{\mathcal{TR}}_{\mathtt{gcd}} + P_{\mathtt{runTime}}, D_0$--- $\vdash$ $(\exists)$ fixpoint $\otimes u(\mathtt{gcd}(0), I)$. Here the post-condition $u(\mathtt{gcd}(0), I)$ forces us to backtrack over all possible execution paths, i.e. CHR derivations, due to non-deterministic constraint and rule selection.

We can reason automatically on the derivation length : There is no CHR derivation of the gcd with only 4 CHR rule applications because the FLORA-2 query "?- $\mathrm{chr}(r_2), \mathrm{chr}(r_2), \mathrm{chr}(r_1), \mathrm{chr}(r_1)$" fails. Similarly, we prove that the gcd can be computed with derivation length 5 and that another CHR derivation with length 8 exists, e.g. $(\underline{24}, 30, \underline{42}) \rightarrowtail_{r_2} (24, \underline{30}, \underline{18}) \rightarrowtail_{r_2} (\underline{24}, 12, \underline{18}) \rightarrowtail_{r_2} (6, \underline{12}, \underline{18}) \rightarrowtail_{r_2} (\underline{6}, \underline{12}, 6) \rightarrowtail_{r_2} (6, \underline{0}, 6) \rightarrowtail_{r_1} (\underline{6}, \underline{6}) \rightarrowtail_{r_2} (6, \underline{0}) \rightarrowtail_{r_1} (6)$.

7 Conclusion

We showed how we can execute and reason on execution of CHR programs within one logical framework by integrating the operational and declarative semantics of CHR into $\mathcal{TR}$. We introduced rule names into the formalism, mapped CHR states and CHR rules to databases and $\mathcal{TR}$ rules, and mapped the CHR run-time system for non-deterministic rule application to a recursively defined $\mathcal{TR}$-predicate. The *perfect-model semantics* of a generalised-Horn $\mathcal{TR}$ program is our new *declarative $\mathcal{TR}$ semantics* of CHR. The model-theoretical executional entailment statement ("one possible execution sequence") brings together $\mathcal{TR}$ program, execution path, and program invocation. The executional deduction inference system mechanically infers a $\mathcal{TR}$-query and computes the necessary updates to the database. We showed execution and automatic reasoning on CHR using the FLORA-2 system.

By bringing the operational semantics of CHR into $\mathcal{TR}$, we merged operational and declarative semantics of CHR in one formal system which allows both execution and reasoning. Our approach is more practical than the one taken for the available declarative semantics of CHR. Both the declarative classic predicate logic semantics and its recent extension to linear logic are more theoretical. They cannot execute a CHR program, cannot reason on its execution, and offer only limited help to mechanise reasoning.

We plan to extend our mapping and to investigate the connections between the formalisms of CHR, constraint programming, and $\mathcal{TR}$ in more detail :

- Lift the restrictions on ground, range-restricted CHR by encoding variables in the database, introduce propagation rules $H \Rightarrow G \mid B$ (which do not remove H upon application), and avoid trivial non-termination, by encoding the *propagation history* in the database.
- Use full $\mathcal{TR}$ to reason on the effect properties [5] of a CHR program starting from our new declarative $\mathcal{TR}$ semantics.
- Another direction is to extend $\mathcal{TR}$ with constraints according to the *general CLP-scheme* [7] which would then allow constraint solving over a side-effect-full, logic programming host language.

As CHR enables the direct implementation of many important monotonic and nonmonotonic reasoning services, this work can be seen as very first step towards a unifying framework to specify, execute, and reason about the semantics of rule-based programs, knowledge bases, and inference engines as envisioned in [10].

Acknowledgements. We thank Thom Frühwirth and the anonymous reviewers for their valuable comments which helped us to improve this paper. Marc Meister and Jacques Robin are supported by the DAAD project ROARS. Khalil Djelloul is funded by the DFG project GLOB-CON.

Références

[1] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2) :133–165, 1999.

[2] H. Betz and T. Frühwirth. A linear-logic semantics for Constraint Handling Rules. In *CP 2005*, volume 3709 of *LNCS*, pages 137–151. Springer, 2005.

[3] A. J. Bonner and M. Kifer. Transaction Logic programming (or, a logic of procedural and declarative knowledge). Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto, Canada, 1995.

[4] A. J. Bonner and M. Kifer. A logic for programming database transactions. In *Logics for Databases and Information Systems*, pages 117–166. Kluwer, 1998.

[5] A. J. Bonner and M. Kifer. Results on reasoning about updates in Transaction Logic. In *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 166–196. Springer, 1998.

[6] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3) :95–138, 1998.

[7] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *J. Logic Programming*, 37(1–3) :1–46, 1998.

[8] M. Kifer. Nonmonotonic reasoning in FLORA-2. In *LPNMR 2005*, volume 3662 of *LNCS*, pages 1–12. Springer, 2005.

[9] M. J. Maher. Logic semantics for a class of committed-choice programs. In *ICLP 87*, pages 858–876. The MIT Press, 1987.

[10] J. Robin. Reused Oriented Automated Reasoning Software (ROARS) project web page, 2005. `http://www.cin.ufpe.br/~jr/mysite/RoarsProject.html`.

[11] T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules. *J. Theory and Practice of Logic Programming*, 6(1&2) :213–224, 2006.

[12] T. Schrijvers et al. The Constraint Handling Rules (CHR) web page, 2007. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.

[13] M. Thielscher. FLUX : A logic programming method for reasoning agents. *J. Theory and Practice of Logic Programming*, 5(4&5) :533–565, 2005.

[14] G. Yang, M. Kifer, C. Zhao, and V. Chowdhary. *FLORA-2 : User's Manual, Version 0.94 (Narumigata)*, 2005. `http://flora.sourceforge.net/docs/`.