CHR 2009

Proceedings of the 6th International Workshop on Constraint Handling Rules

Editors: Frank Raiser & Jon Sneyers

July 15th, 2009
Pasadena, California, USA
at the occasion of ICLP 2009

Preface

This volume contains the papers presented at CHR 2009, the sixth international workshop on Constraint Handling Rules, held on July 15th, 2009 at the occasion of ICLP 2009 and IJCAI 2009 in Pasadena (California, USA).

Previous workshops on Constraint Handling Rules were organized in May 2004 in Ulm (Germany), in October 2005 in Sitges (Spain) at ICLP, in July 2006 in Venice (Italy) at ICALP, in September 2007 in Porto (Portgual) at ICLP, and in July 2008 in Hagenberg (Austria) at RTA. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments.

The Constraint Handling Rules (CHR) language has become a major declarative specification formalism and implementation language for constraint reasoning algorithms and applications. Algorithms are often specified using inference rules, rewrite rules, sequents, proof rules, or logical axioms that can be directly written in CHR. Its clean semantics facilitates program design, analysis, and transformation.

This edition of the CHR workshop received nine full paper submissions. Each submission was carefully reviewed by three referees. The program committee maintained a very high standard of quality and decided to accept six papers.

The program also includes an invited talk by Thom Frühwirth and a discussion session about future research directions for CHR. Finally, we highly recommend the talk by Mike Elston at the CULP workshop (Commercial Users of Logic Programming), which takes place the day after this CHR workshop.

We are grateful to all the authors of the submitted papers, the program committee members, and the referees for their time and efforts spent in the reviewing process. We also thank the ICLP 2009 organizers, in particular the ICLP workshop chair Manuel Carro, for hosting our workshop.

July 2009

Frank Raiser
Jon Sneyers

Organization

Workshop Coordinators:

- Frank Raiser (Germany)
- Jon Sneyers (Belgium)

Program Committee:

- Hariolf Betz, Universität Ulm (Germany)
- Henning Christiansen, Roskilde University (Denmark)
- Leslie De Koninck, Katholieke Universiteit Leuven (Belgium)
- François Fages, INRIA Rocquencourt (France)
- Thom Frühwirth, Universität Ulm (Germany)
- Rémy Haemmerlé, Universidad Politécnica de Madrid (Spain)
- Edmund Lam, National University of Singapore (Singapore)
- Eric Monfroy, Université de Nantes (France)
- Frank Raiser, Universität Ulm (Germany)
- Jacques Robin, Universidade Federal de Pernambuco (Brazil)
- Beata Sarna-Sarosta, LogicBlox Inc., Atlanta (USA)
- Tom Schrijvers, Katholieke Universiteit Leuven (Belgium)
- Jon Sneyers, Katholieke Universiteit Leuven (Belgium)
- Peter Stuckey, NICTA Victoria Laboratory (Australia)

Referees:

Hariolf Betz	Henning Christiansen	Leslie De Koninck
François Fages	Thom Frühwirth	Rémy Haemmerlé
Edmund Lam	Eric Monfroy	Frank Raiser
Jacques Robin	Beata Sarna-Sarosta	Tom Schrijvers
Jon Sneyers	Peter Stuckey	Peter Van Weert

Table of Contents

Invited talk: First steps towards a lingua franca for computer science: Rule-based Approaches in CHR 1
Thom Frühwirth

Recommended external talk: From Prolog to Porsche — Experiences developing a large scale financial application in Prolog 2
Mike Elston

Extending CHR with objects under a variety of inheritance and world-closure assumptions 3
Marcos Aurelio Almeida da Silva, Jacques Robin

On connections between CHR and LCC. 18
Thierry Martinez

Equivalence of CHR States Revisited 34
Frank Raiser, Hariolf Betz, Thom Fruehwirth

Operational Equivalence of Graph Transformation Systems 49
Frank Raiser, Thom Fruehwirth

CHRiSM: Chance Rules induce Statistical Models 62
Jon Sneyers, Wannes Meert, Joost Vennekens

A Proposal for a Next Generation of CHR 77
Peter Van Weert, Leslie De Koninck, Jon Sneyers

First steps towards a lingua franca for computer science: Rule-based Approaches in CHR

Invited talk

Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. The notion of "rule" is ubiqitous in computer science, from theoretical formalisms to practical programming languages. Matured rule-based programming experiences a renaissance due to its applications in areas such as business rules, semantic web, computational biology, medical diagnosis, software verification, and security.
We will embed rule-based approaches into the Constraint Handling Rules (CHR) language by simple source-to-source transformations. We also consider the other direction of embeddings. This gives us the possibility to compare and analyse the different approaches.

- Classical Rule-Based Systems
  - Production Rule Systems
  - Event-Condition-Action Rules
- Rewriting- and Graph-based Formalisms
  - Term Rewriting Systems and Graph Transformation Systems
  - Chemical Abstract Machine and Multiset Transformation
  - Petri Nets
- Constraint- and Logic-based Programming
  - Deductive Databases and Logical Algorithms
  - Prolog and Constraint Logic Programming
  - Concurrent Constraint Programming

In this talk, we will highlight the commonalities of the transformations and what of their features may suggest themselves as extensions of CHR — as indeed they have already been proposed in the literature:

- rule priorities
- negation-as-absence
- disjunction and search
- diagrammatic notation

On the other hand, a simple fragment of CHR can cover rule-based approaches without constraints.

From Prolog to Porsche

Experiences developing a large scale financial application in Prolog

Mike Elston

Scientific Software and Systems Ltd, New Zealand

Abstract. Work on the Y2K bug ten years ago exposed widespread dissatisfaction with the software applications available to New Zealand stock brokers. Sensing an opportunity and having successfully applied Prolog to optimization problems in agriculture, Scientific Software and Systems Ltd developed SecuritEase, a comprehensive stock broking dealing and settlement system. We quickly determined that the main problem with the incumbent systems was their lack of flexibility and consequent inability to accommodate the demands of increasing competition and globalization. Noting that the response to these challenges by our competitors was to outsource development to lower wage economies, SSS resolved instead to use Prolog to increase development productivity through the creation of a set of domain specific tools. Using these tools, SSS analysts and programmers worked together to create, refine and deploy the SecuritEase system. Within five years SSS had displaced its competitors to become the dominant supplier in New Zealand. Many lessons were learned on the way including: the importance of Microsoft Windows and just how difficult it is to achieve 24x7 reliability. This paper describes how Prolog allowed the development of three key tools: a forward chaining user interface system, a productive and efficient relational database interface, and an approach to quickly building financial messaging interfaces. It also describes how the many aspects of a large scale Prolog-based project were managed including finding, training and retaining staff, achieving reliability, Prolog performance, the role of open source software, the need for constant refactoring, and the benefits of a single language approach. Finally it describes how the application is now being adopted by much larger companies outside New Zealand and the particular challenges associated with justifying Prolog to a sceptical audience.

This talk is part of the program of the "Commercial Users of Logic Programming" workshop at ICLP 2009 (July 16th).

More information about the CULP workshop:
`http://www.cs.kuleuven.be/~toms/CULP2009/`

Extending CHR with objects under a variety of inheritance and world-closure assumptions

Marcos Aurélio A. da Silva[1], Jacques Robin[1]

Centro de Informática – Universidade Federal de Pernambuco
Recife – PE – Brazil {maurelio1234, robin.jacques}@gmail.com

Abstract. In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) an Object-Oriented (OO) extension of CHRD. Syntactically, CHORD integrates two versatile dual programming and knowledge representation languages: Flora, a hybrid OO rule-based language and CHRD. A CHORD program extends CHRD by allowing Flora-style object constraints (o-constraints) in the head, guard or body of its rules. Orthogonal to its rules, a CHORD program also contains semantic assumption directives, an innovative construct that maximizes semantic versatility. Each directive defines a point along one dimension of the space of semantic assumptions made by various OO and rule-based languages in order to complete the knowledge explicitly specified in them by complementary knowledge left implicit by the programmer. Among others, these assumptions specify what kind of world closure and inheritance is desired.

1 Introduction

In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) the first bona-fide Object-Oriented (OO) rule-based constraint programming language. CHORD integrates most advanced features of the OO paradigm with those of the rule-based and constraint based paradigms. It thus aims at bringing together within a single language the well-known benefits of objects for programming-in-the-large within a systematic software engineering process, with the unique facilities of rules and constraints for programming-in-the-small intelligent applications that require advanced embedded automated reasoning services. Fast prototyping such applications requires a language that supports not only Turing-complete programming but also formally well-founded declarative Knowledge Representation (KR).

Syntactically, CHORD integrates the cores of two exceptionally versatile dual programming-KR languages: Flora [14] a hybrid OO rule-based language and Constraint Handling Rules with Disjunctions (CHRD) [1], a hybrid rule and constraint based language. A CHORD program extends CHRD by allowing Flora-style object constraints (o-constraints) in the head, guard or body of its rules, Logically, these o-constraints are conjunctions of atomic constraints, each one defining or referencing one structural or behavioral feature of a class or object.

Orthogonal to its rules, a CHORD program also contains *semantic assumption directives*. This innovative construct maximizes the semantic versatility. Each directive defines a point along one dimension of the space of semantic assumptions made by various OO and rule-based languages in order to complete the knowledge explicitly specified in them by complementary knowledge left implicit by the programmer. The combination of leaving semantics largely orthogonal to syntax, with the integration of three powerful declarative paradigms, allows CHORD to elegantly encode and unify programs and knowledge bases from a wide variety of programming and KR languages based on objects and/or constraints and/or rules.

Compared to imperative OO languages such as Java, C#, C++ or Python, CHORD presents the advantage of being a dual programming and KR language that provide as built-in, within its execution platform, a powerful, general purpose rule-based constraint solving inference service. This allows fast prototyping intelligent systems. It also presents the advantage of possessing a simple formal declarative semantics in Classical First-Order Logic (CFOL), which allows to directly use consolidated theorem proving technology to verify properties of a CHORD implementation. Compared to Flora and other OO rule-based languages such as JESS [5] and ILOG Rules[1], CHORD presents the advantage of possessing a configurable semantics that can support various inheritance strategies together with not only the fully closed-world assumption, but also the selectively, partially closed-world assumption or the fully open-world assumption [11].

The rest of the paper is organized as follows. In Section 2, we overview CHRD, which rules, relational constraints, declarative CFOL semantics and operational semantics are largely reused by CHORD. In Section 3, we show how CHORD's syntax extend CHRD's syntax with Flora-style objects. In Section 4, we describe the semantic assumption space covered by CHORD together with the directives used in a CHORD program to choose a point in this space that indicate to the CHORD engine which semantic interpretation to choose for the OO rules of the program. In Section 5, we discuss the current CHORD engine implementation which translates a CHORD program into a semantically equivalent CHRD program which can in turn be executed on any CHRD platform such as SWI-Prolog. In Section 6, we compare CHORD with closely related language families, namely (a) dual programming and KR OO rule-based languages with a formal semantics and (b) programming languages that pioneered the integration of objects with rules and constraints. In Section 7, we conclude by highlighting the contributions of our research and discuss its possible future directions.

2 Introduction to CHRD

A CHRD rule base is composed by a set of CHRD rules. The following CHRD rule defines the `append(X,Y,Z)` constraint:

[1] ILOG Business Rules Management Systems:
`http://www.ilog.com/products/businessrules/index.cfm`

```
r1 @ append(X,Y,Z) <=>
(X = [], Z = Y);(X = [H|L1], Z = [H|L2], append(L1,Y,L2)).
```

In this rule, `Z` is a list composed by the elements of the list `X` followed by the elements of the list `Y`. If `append(X,Y,Z)` holds, we have two options: (i) `X=[]` and, therefore, `Z=Y`; or (ii) `X` is a list in the form `[H|L1]`, and thus, `Z` is composed by `H` followed by `L1` and then followed by the elements in `Y`.

There are three kinds of rules in CHRD: simplification, propagation and simpagation. The simpagation rules are the most general category of rules and they have the following form $r@H_k \backslash H_r \Leftrightarrow G|B.$, where `r` is an identifier for the rule, H_r and H_k are the heads of the rule, G is the guard and B is the body. If the guard is `true`, it can be omitted. The operational semantics of such rule is that if H_k and H_r are found in the constraint store and the guard G is entailed by it, the constraints in H_r should be removed and the constraints in the body B should be added to the constraint store. If H_k is empty, this rule is called a *Simplification Rule*, and this part of the rule is omitted. On the other side, if H_r is empty, this rule is called a *Propagation Rule*. In this case, the second part of the head of the rule is omitted and the $\Leftrightarrow$ is replaced by the symbol $\Rightarrow$.

The declarative semantics of a rule base is the conjunction of the declarative semantics of each rule. For each kind of rule, its abstract syntax and its declarative and operational semantics are described in the following table:

Rule	Abstract Syntax	Declarative Semantics	Operational Semantics
Simplification	$r@H \Leftrightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H \leftrightarrow \exists \bar{y} B))$	$\langle H \rangle \mapsto_G \langle B \rangle$
Propagation	$r@H \Rightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H \rightarrow \exists \bar{y} B))$	$\langle H \rangle \mapsto_G \langle H, B \rangle$
Simpagation	$r@H_k \backslash H_R \Leftrightarrow G\|B$	$\forall \bar{x}(G \rightarrow (H_k \wedge H_R \leftrightarrow \exists \bar{y} H_1 \wedge B))$	$\langle H_k, H_R \rangle \mapsto_G \langle H_k, B \rangle$

In this table $\bar{y}$ denotes the set of variables that appear in the rule body B but do not appear anywhere else in the rule and $\bar{x}$ the set of the remaining variables.

Example 1. Let us suppose that the current state of the constraint store is $append([1],[2],Z)$. The actual execution of this rule base is represented by the following set of transitions:

$$\begin{aligned}
&\langle append([1],[2],Z) \rangle \mapsto_{r_1} \\
\langle [1] = [], Z = [2] \rangle \,|\, &\langle [1] = [1|[]], Z = [1|L2], append([],[2],L2) \rangle \mapsto_* \\
&\langle [1] = [1|[]], Z = [1|L2], append([],[2],L2) \rangle \mapsto_{r_1} \\
&\langle [1] = [1|[]], Z = [1|L2], [] = [], L2 = [2] \rangle \\
|\, \langle [] = [H|L1], L2 = [H|L2'], append(L1,[2],L2') \rangle &\mapsto_* \\
&\langle [1] = [1|[]], Z = [1|L2], [] = [], L2 = [2] \rangle
\end{aligned}$$

The notation $\langle G_0 \rangle \,|\, \ldots \,|\, \langle G_n \rangle$ represents the alternative execution states. The transition $\mapsto_{r_1}$ represents the application of r_1 and the transition $\mapsto_*$ the removal

of failed states. In the initial state we add the constraint $append([1],[2],Z)$ to the initial constraint store, the rule r_1 is applied and then we get two alternative execution states, notice that the first one is failed because of the constraint $[1] = []$. The next step removes this failed execution state. The next step applies the rule r_1 to the constraint $append([],[2],L2)$ and we obtain again two alternative execution states. This time, the second one is failed because of the constraint $[] = [H|L1]$. The next step removes this failed state and we get to the final state, where no other rule is applicable. From this final state we can thus conclude $Z = [1,2]$.

3 The Syntax of CHORD

CHORD extends CHRD with a set of predefined *object-oriented* constraints (the so-called *o-constraints*) that appear in the rule heads and in the rule bodies. There are essentially 4 kinds of o-constraints:

- $o : c$, whose meaning is: the object o is an instance of the class c.
- $s :: g$, whose meaning is: the class s is a subclass of the class g.
- $o[a = v]$, whose meaning is: the value of the attribute a of the object o is v.
- $c[a *= v]$, whose meaning is: the value of the attribute a to be inherited by instances of the class c is v.

This extension is called *Core CHORD*. On top of it we build *Full CHORD* which extends first one with syntactic sugar that is translated into *Core CHORD* by the compiler. The first set of syntactic sugar is the *o-molecules* that are sets of atomic o-constraints condensed into only one constraint like `o:c[a=t,b*=u]` that is represented in *Core CHORD* by the conjunction `o:c, o[a=t], o[b*=u]` Another sort of syntactic sugar is the path expressions. They appear as values inside constraints in a CHORD program like in `p(a.b.c.d)` and they can be translated into conjunctions of constraints such as `p(D), a[b=B], B[c=C], C[d=D]`.

A CHORD program is a set of rules annotated with at most *one* semantic directive (for the whole program), that defines the semantics of its o-constraints. This directive has the following syntax:

```
semantics [
        s1 [ p11, ..., p1n ],
        ...,
        sm [ pm1, ..., pmn ]
  ].
```

Notice that semantic assumptions may optionally have parameters. Let us take the following directive as an example:

```
semantics [
       simpleInheritance,
       classes [a, b, c]
].
```

It specifies that in the annotated CHORD rule base: (i) only *simple inheritance* is allowed, i.e., every class must have at most one direct superclass; and (ii) there are only three classes: a, b and c and no other classes.

4 The Semantic Assumptions Taxonomy

In this Section, we propose a general ontology for the implicit semantics assumptions of knowledge representation and programming languages that include constructs for classes and objects, optionally integrated with constructs for constraints and rules. We present it completely in an intuitive manner as UML[9] diagrams and natural language definitions.

To illustrate our ontology, we consider the knowledge base shown in Fig. 1 as UML class and object diagrams annotated with OCL[10] constraints. In this example we define the class `GroupMember` with two attributes: `pacifist`, a boolean and `color` a string. It has two subclasses: `ReligionMember` and `PartyMember` which redefine the attributes defined in the superclass.

The `ReligionMember` class redefines `pacifist` as being derived from the `pacifist` attribute of the `Religion` of the member. The `PartyMember` redefines `pacifist` as being derived from the `pacifist` attribute of the `Party` of the member. They both redefine `color` and set it to `'red'` by default. The class `President` is a subclass of both `ReligionMember` and `PartyMember` and redefines its attributes. It defines two OCL invariants: (i) the `pacifist` attribute should be equal to `party.pacifist` and (ii) the `oppositeParty` should not be the same as the `party`. It also defines the default value of `color` to `blue`.

We define four instances in our model. `nixon` is a `President` with `religion` set to `quaker` (which is pacifist) and `party` set to `republican` (which is not pacifist).It also sets the value of the attribute `impeached` as `true`. We also have the instance `john` of the class `Vice`. Note that we slightly abuse the UML/OCL notation to provide a visual rendering of an example that covers constructs covered that are not acceptable in UML/OCL. In particular, the definition of the `impeached` feature in the nixon object instance of a class where such feature is not defined is not allowed in UML/OCL. Similarly for the definition of the `john` as an instance of a non-defined class `Vice`.

In Fig. 2, we display the overview of our space of semantic assumptions. We divide assumptions space into two orthogonal dimensions: $WorldAssumption$ and $InheritanceAssumption$.

By $WorldAssumption$, we mean the distinction between closed and open world [11]. More specifically, we focus on the *closure* of the set of classes, instances and features of a language. In a language where $closedClasses = true$ the `john` instance would be valid only if the `Vice` class was declared. In a language where $closedInstances = true$, the OCL invariant on the attribute `oppositeParty` of the class `President` would not be valid, since there is only one declared instance of `Party`. In a language with $closedInstanceFeatures = true$, the slot `impeached` of the instance `nixon` would be forbidden, since it is not declared in the class diagram. In a language with $closedClassFeatures = false$,

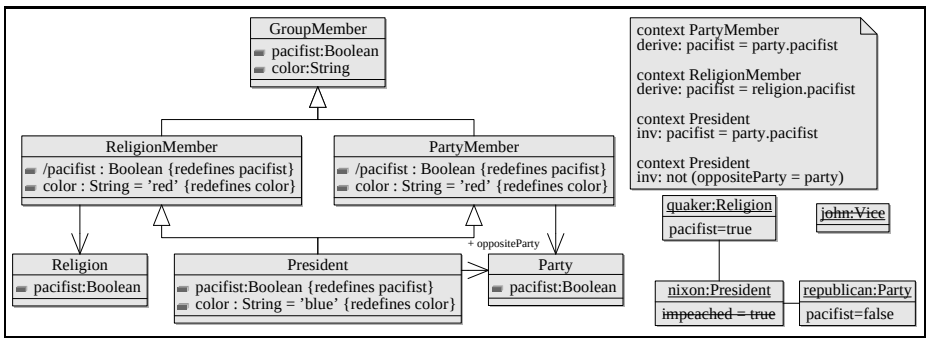

Fig. 1. UML Example

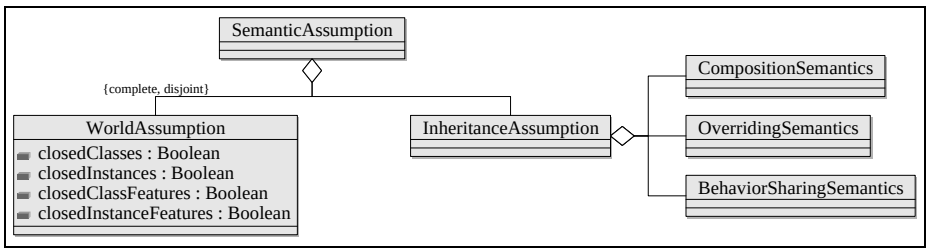

Fig. 2. Semantic Assumptions: Overview

an inference engine would use the slot `impeached` in `nixon` to deduce that some of its superclasses defines this attribute, even though it is not made explicit in the diagram.

The other dimension we explore in our ontology is the *InheritanceAssumption*, that we have divided into three sub-dimensions: *CompositionSemantics*, *OverridingSemantics* and *BehaviorSharingSemantics*, which are detailed in Fig. 3.

The *CompositionSemantics* (on the left side in Fig. 3) defines the way classes can be connected by the means of the subclass relationship. It is partitioned into two kinds of *CompositionSemantics*: the *SimpleInheritance* and the $MultipleInheritance$. as indicated by the UML constraints {*disjoint, complete*}). In the first case, we allow each class to be a direct subclass of only one other class. In the second one, there is no such restriction. In the example in Fig. 1, the class `President` would be invalid if we have a language with *SinpleInheritance*.

In case of $MultipleInheritance$, we need to specify a strategy to resolve potential conflicts, which may arise when the same feature is defined in several superclasses with incompatible signatures or values. In our ontology, we divide $ConflictResolutionStrategy$ into four disjoint subclasses: the $ValueBased$, the *SourceBased*, the *PriorityBased* and the $ExplicitDefinitionBased$. These

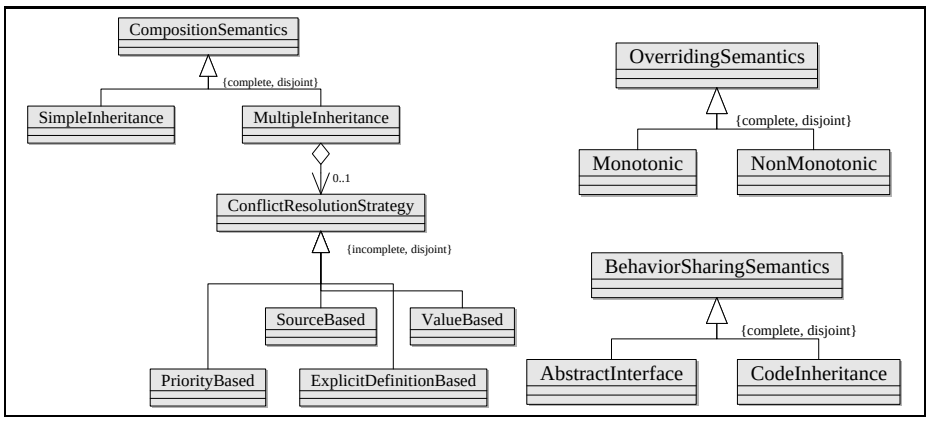

Fig. 3. Inheritance Assumptions: Composition, Overriding and Sharing Semantics

subclasses form are incomplete division in the sense that other strategies, not covered here, may be used by other languages.

In the example in Fig. 1, in the class `President` the features `color` and `pacifist` conflict. In the $ValueBased$, if the multiple feature definitions provide the same *value* the inheritance is allowed. In our example, the `President` class would inherit $color = red$ if this feature were not redefined in this class. In the $SourceBased$, the inheritance is always denied if there is a conflict.

In the $PriorityBased$ the source is chosen by the means of a priority between the conflicting superclasses. In our example, the `pacifist` from the class `ReligionMember` could be preferred to the one of the class `PartyMember`. In the $ExplicitDefinitionBased$, the user is responsible for designing a model without conflicts. In our example, UML forces the class `President` to override the conflicting attributes of its superclasses.

The $OverridingSemantics$ (on the center in Fig. 3) defines whether the *overriding* is allowed ($NonMonotonic$) or not ($Monotonic$). In languages with classical FOL semantics, *overriding* is usually not allowed. In our example, the features containing the keyword `redefines` are, in fact, overriding the feature of same name in the superclass.

The $BehaviorSharingSemantics$ (on the right side in Fig. 3) defines how features are shared with their subclasses. In the $AbstractInterfaceInheritance$, only the feature signature is inherited, whereas in the $CodeInheritance$, the feature signature and code are both inherited by the subclasses.

In the next table we map every class in our ontology into its concrete syntax in the semantic directive to be used in a CHORD program. The assumptions marked with a "–" are used by default.

Assumption		Syntax
Composition Semantics	SimpleInheritance	`simpleInheritance`
	MultipleInheritance	–
	ValueBased	`valueBasedConflictResolution`
	SourceBased	`sourceBasedConflictResolution`
	PriorityBased	`priorityBasedConflictResolution`
	ExplicitDefinitionBased	`explicitConflictResolution`
Overriding Semantics	Monotonic	`noOverriding`
	NonMonotonic	–
BehaviorSharing Semantics	AbstractInterface	–
	CodeInheritance	`code[(c,[f*])*]`

WorldAssumption		`classes [c*]`
		`classesHierarchy [(s,g)*]`
		`objects [o*]`
		`classFeatures [(c,[f*])*]`
		`objectFeatures [(o,[f*])*]`
		`localValues [(o,[f*])*]`
		`inheritables [(c,[f*])*]`

Some assumptions require a list of parameters to be provided. The *CodeInheritance* requires the user to define the set of features to which the code inheritance should be enabled. The list of possibilities in the *WorldAssumption* allow the user to selectively close the world for part of its model. The *classes* directive closed the set of classes, the *classesHierarchy* directive closes the graph formed by the hierarchy of classes in the model, the *objects* directive closes the set of objects in the model, the *classFeatures* and the *objectFeatures* directives close the set of features (operations and attributes) in an object or in a class, the *localValues* directive closes the set of features that are defined locally in the model (i.e. that should not be defined by inheritance) and the *inheritables* directive closes the set of features that are marked as inheritable[2] .

5 The CHORD Inference Engine

In this Section we analyze the implementation of an inference engine for CHORD on top of the SWI Prolog 5.6.61 and that is available in `http://chord.sf.net/`. It consists of approximately 1500 lines of Prolog and CHRD code, including the code for the trailing rules. The inference engine is composed by a Runtime and a Compiler. The function of the Runtime is to extend the SWI Prolog default console runtime with special commands to interact with the CHORD inference

[2] At first sight the assumption directives for partially closing the world may seem to verbosely declare facts that could be automatically be infer from the rules in the program, but remember that in some programs part of the model may exist that are not referred in the program rules.

engine, i. e., for compiling CHORD programs, running the engine with a provided goal, providing help and etc.

The function of the Compiler is to read CHORD rule bases and generate the equivalent CHRD rule base for them, this final rule base is obtained by:

1. Translating *Full CHORD* into *Core CHORD*.
2. Interpreting every o-constraint as a user-defined constraint, e.g., the constraint $a : b$ is interpreted as the binary constraint $: (a, b)$.
3. Translating each assumption in the semantic directive into a CHRD rule base (notice that parameterless assumptions are translated by adding a fixed set of rules to the final rule base, but the other ones will generate a different set of rule depending on the parameters). This set of rules is called *Trailer Rules*.
4. Appending the set of Core Rules.

The Core Rules specify the semantics of the o-constraints in a way that is not dependent on the semantic assumptions. For example, the following rules are part of the core rules:

```
% 1. taxonomy transitivity
A::B, B::C ==> A::C.
A:B,  B::C ==> A:C.

% 2. monotonic feature inheritance
O:C, C[F*= V] ==> O[F=W].
```

The first group of rules defines the transitivity of the $:: (\cdot, \cdot)$ and of the $: (\cdot, \cdot)$ relationships and the second one defines that if O is an object of a class C that defines an inheritable value V for the feature F, then there exists a value W for this feature in O. The value of such feature depends on the semantic assumptions, e.g. this inheritable value may be *overridden* by a more specific superclass of O.

To illustrate the Trailer Rules, we present the translation of the `simpleInheritance` assumption used in the previous example:

```
X::Y, X::Z ==> (Y::Z ∨ Z::Y).
X:Y,  X:Z  ==> (Y::Z ∨ Z::Y).
```

These rules state that if a class or an object has two superclasses these superclasses should be related, i.e., one of them should be more specific than the other, avoiding multiple inheritance.

Example 2. Let us consider the following CHORD program:

```
semantics [ ].
facts <=> nixon : republican.
```

In this program we define the object `nixon` and the class `republican`. We say that `nixon` is an instance of `republican`. If we run this program with the `facts, nixon : party` initial constraint store, we are going to get only one final store, with the following state:

```
nixon : republican, nixon : party
```

If we change the `simpleInheritance` semantics directive and run this example with the same initial constraint store we are going to get three final states:

```
S1: nixon : republican, republican :: party
S2: nixon : party, party :: republican
S3: nixon : party, republican :: party, party :: republican
```

Notice that assuming the simple inheritance changed our final state. Since we added two classes to `nixon` in the constraint store, there are only three ways to restore its consistency:

1. Assume that `party` is a super class of `republican`
2. Assume that `party` is a sub class of `republican`
3. Assume that `party` and `republican` are the same class by stating that one is a subclass of the other

6 Related Work

6.1 OO Rule based Languages

F-logic. F-logic [14] is an extension of Prolog with object-oriented predicates on top of Well-Founded Models [6] with Negation-As-Failure (NAF)[7] to represent the default reasoning involved in inheritance. In terms of our Semantic Assumptions Ontology, F-logic employs the Closed World Assumption with Multiple Inheritance using Source Based Conflict Resolution Strategy, Overriding and Code Inheritance.

A rule has a head and a body, with the following concrete syntax: "`head :- body.`". A rule with a "`true`" body can be written as "`head.`" and it is called a *fact*. The body of a rule may be represented by a conjunction of formulas with `f0,...,fn` as concrete syntax. F-logic extends the set of terms with the so-called F-Atoms and the F-Molecules. The first group contains terms that express *atomic* informations about the object model, e.g, `O:C` means `O` is an instance of `C`, `C::S` means `C` is a subclass of `S`, `O[F->V]` means `V` is the value of the feature `F` in the object `O` and `C[F*->V]` means `V` is the value of the feature `F` to be inherited by instances of the class `C`.

The following example illustrates how we can represent F-logic Knowledge Bases in CHORD.

Example 3. Let us consider the following F-logic program:

```
nixon:quaker.
nixon:republican.
quaker[policy->pacifist].
republican[policy->hawk].
```

It defines two classes for the `nixon` object: `quaker` and `republican` which define different inheritable values for the `policy` feature: `quaker` defines its `policy` to `pacifist` and `republican` defines its `policy` to `hawk`. In this case, `nixon` should inherit neither `pacifist` nor `hawk` for `policy`, since there is a conflict between its subclasses.

The CHORD rule base that implements this F-logic program follows:

```
% semantic assumptions:
%         closed world,
%         multiple source based inheritance
semantics [
        overriding,
        sourceBasedMultipleInheritance,
        objects [nixon],
        classes [quaker, republican],
        classesHierarchy [ ],
        localValues [ (nixon, []) ]
].

facts <=> nixon : quaker, republican[policy *= hawk],
          nixon : republican, quaker[policy *= pacifist].
```

The final constraint store for an initial constraint store containing only the `facts` constraint is:

```
quaker[policy*=pacifist], nixon[policy=_]
republican[policy*=hawk],
nixon:republican
nixon:quaker
```

The value for the `policy` feature of `nixon` remains undefined agreeing with the expected F-logic semantics of this rule base: the inheritable values defined by its superclasses for this feature conflict and therefore nixon should not inherit any value for it.

The equivalence of the translated CHORD rule base is given in terms of the three-valued object model represented by the F-logic rule base: every *positive* fact in the F-logic model, should be entailed by the FOL semantics of the translated CHRD rule base and every *negative* fact should turn it into an inconsistent base. For example, the initial constraint store `facts, nixon:X, X::republican` (i.e., exists `X` such that `nixon` is an instance of `X` and `X` is a subclass of `republican`) has no consistent final store, because according to the closed world assumption held by the rule base no such `X` class can exist.

Implementing F-logic's inheritance on top of CHRD is not so innovative as Kaeser & Meister have already tried to do it in [8]. In a few words, they implemented an inference engine for a subset of F-logic on top of CHRD forward propagation in order to compute the intended object model. When compared

to our work, their work just shows the feasibility of the idea while failing in providing a correct account for one of the most complex features of the language: the interaction between inheritance and deduction. In the present work, we do not intend to provide an inference engine for F-logic, but we want to provide F-logic's object-oriented capabilities to CHRD programmers.

6.2 OO Constraint Programming Languages

LAURE. LAURE [2] was developed in 1988 at Bellcore Labs. It aims to be used in designing *complex applications* in which AI needs to be embedded into the software. Its semantics is based on a CFOL sublanguage with equality. It distinguishes between objects (whose equality is based on object identity) and values (whose equality is based on equality of feature values) and supports only binary relations.

In terms of our Semantic Assumptions Ontology, LAURE employs the Open World Assumption with Multiple Inheritance and no Code Inheritance. Since there are no inheritable vales there are no conflicts (and thus no conflict resolution Strategy) and no overriding. We are going to use the following example to demonstrate the use of OO concepts in LAURE and how to translate them into CHORD:

Example 4. Let us consider this piece of code that is part of a constraint solver for the problem of automatic graphical layout:

```
% declaration 1
[define RECTANGLE class superclass {GRAPHICAL_OBJECT}]

% declaration 2
[define Xul attribute domain RECTANGLE,
        -> {0 -- MaxX},
        comment "x-coordinate of the upper left corner"]

% declaration 3
[define Yul attribute domain RECTANGLE,
        -> {0 -- MaxY},
        comment "Y-coordinate of the upper left corner"]
...

% declaration 4
[define DISJOINT class superclass {object},
          with   slot( r1 -> RECTANGLE),
                 slot( r2 -> RECTANGLE)]

% declaration 5
[define disjoint_spec constraint
     for_all (D DISJOINT),
```

```
if   xul1 = Xul(r1(D)), xdr1 = Xdr(r1(D)),
     yul1 = Yul(r1(D)), ydr1 = Ydr(r1(D),
     xul2 = Xul(r2(D)), xdr2 = Xdr(r2(D)),
     yul2 = Yul(r2(D)), ydr2 = Ydr(r2(D)),
    then      [or ydr2 > yul1, ydr1 > yul2,
                  xdr1 < xul2, xdr2 < xul1]]
```

This code is divided into four declarations. The first one defines the `RECTANGLE` class as a subclass of `GRAPHICAL_OBJECT`. The second one defines `Xul` (x-coordinate of the upper left corner) as an attribute of the class `RECTANGLE` with range going from `0` to `MaxX` and the third one defines `Yul` (x-coordinate of the upper left corner) as an attribute of the same class with range going from `0` to `MaxY` (the definition of the coordinates of the other corners was removed from this example. The fourth declaration defines the `DISJOINT` class that associates, by the means of its slots, two different rectangles. Finally, the last declaration provides the semantics for the `DISJOINT` class as a constraint defining that if two rectangles are disjoint their areas should not overlap.

This LAURE program can be translated into the following CHORD program:

```
% semantic assumption: open world without overriding
semantics [ noOverriding ].

% translated declaration 1
facts ==> rectangle :: graphicalObject.

% translated declaration 2
X:rectangle[xul=Xul] ==> Xul > 0, Xul <= maxX.

% translated declaration 3
X:rectangle[yul=Yul] ==> Yul > 0, Yul <= maxY.
...

% translated declaration 4
facts ==> disjoint::object.
X:disjoint[r1=R] ==> R:rectangle.
X:disjoint[r2=R] ==> R:rectangle.

% translated declaration 5
D:disjoint ==> (D.r2.ydr > D.r2.yul, D.r1.ydr > D.r2.yul) ;
               (D.r1.xdr < D.r2.xul, D.r2.xdr < D.r1.xul).
```

Oz. The Oz [12] language was developed in the early 1990s at DFKI with the objective of being a high-level concurrent constraint programming language to overcome the problems in concurrent object oriented languages at the time. It borrows ideas from logic programming (such as logic variables and data structures), and from concurrent programming (such as cells). When compared to

CHORD's model of inheritance, Oz's is very much restricted. It is based based on a very procedural approach with low level hard-to-learn concepts.

7 Conclusion

In this paper, we present CHORD (Constraint Handling Object-oriented Rules with Disjunction) the first bona-fide Object-Oriented (OO) rule-based constraint programming language. CHORD integrates most of the advanced features of the OO paradigm with those of the rule-based and constraint-based paradigms. It thus brings together within a single language the well-known benefits of objects for programming-in-the-large within a systematic software engineering process, with the unique facilities of rules and constraints for programming-in-the-small intelligent applications that require advanced embedded automated reasoning services.

The main innovative distinctive feature of CHORD over previous programming and KR languages that integrate objects with rules and/or constraints is that it allows the programmer to piecemeal choose, via header directives, which semantic assumption the CHORD engine should make about inheritance as well as the closeness of the universe of class names, class hierarchies, object names, class and object feature names and values. For a given CHORD OO rule base O and two different semantic assumption directives A1 and A2, the CHORD engine translates the CHORD program into distinct CHRD bases R1 and R2 before running them on the underlying CHRD engine that it relies on for constraint solving.

This makes CHORD uniquely versatile in terms of potential application fields. It can be used as a declarative programming language to implement constraint solvers in taxonomically rich domains. In can also be used as a knowledge representation language for semantic web ontologies, services and agents and expert systems in taxonomically rich domains. It also has potential as a model transformation language for model-driven engineering [13].

In future work, we intend to empirically evaluate the benefits of CHORD as compared to standard languages on both benchmark and real-world programs and knowledge bases in each of these application domains. We also intend to integrate CHORD with software components, one key programming-in-the-large feature that it still misses. This integration will feature additional constructs to encapsulate CHORD bases into components that can be assembled by connecting their provided and required services interfaces. One source of inspiration for this future integration is CHR^{at} which extends CHR with modules and rule-based deep guard entailment [4].

Remark. An extended version of this work can be found in [3], including complete source code of the Core and Trailer Rules and an in-depth description of the CHORD inference engine and of the case studies presented here (along with other ones).

References

1. Slim Abdennadher. A Language for Experimenting with Declarative Paradigms. *Second Workshop on Rule-Based Constraint Reasoning and Programming*, 2000.
2. Yves Caseau. Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *JAIPS*, 1993.
3. Marcos Aurélio Almeida da Silva. CHORD: Constraint Handling Object-oriented Rules with Disjunctions. Master's thesis, Universidade Federal de Pernambuco, February 2009. Universidade Federal de Pernambuco.
4. François Fages, Cleyton Mario de Oliveira Rodrigues, and Thrierry Martinez. Modular CHR with ask and tell. *Proceedings of the Fifth Workshop on Constraint Handling Rules (CHR 2008)*, 2008.
5. Ernert Friedman-Hill. *JESS in Action*. Manning Publications, 2003.
6. A. V. Gelder, K. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of ACM*, 38(3):620–650, 1998.
7. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.
8. Martin Kaeser and Marc Meister. Implementation of a F-logic Kernel in CHR. *Third Workshop on Constraint Handling Rules*, pages 33–48, 2006.
9. OMG. Unified Modeling Language (UML), Version 2.0. Technical report, Object Management Group Inc, May 2004.
10. OMG. OCL 2.0 Specification. Technical report, Object Management Group Inc, June 2005.
11. Raymond Reiter. *On Closed World Databases*. In Gallaire, H. and Minker, J., editors, Logic and Databases. Plenum Press, New York, 1978.
12. Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in oz. In Pascal Van Hentenryck Vijay Saraswat, editor, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, Cambridge, MA, May 1995.
13. Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
14. Guizhen Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases*. PhD thesis, Stony Brook University, December 2002.

On connections between CHR and LCC

Thierry Martinez

Contraintes Project–Team, INRIA Paris–Rocquencourt, France

Abstract. Both CHR and LCC languages are based on the same model of concurrent computation, where agents communicate through a shared constraint store, with a synchronization mechanism based on constraint entailment. The *Constraint Simplification Rules* (CSR) subset of CHR and the flat subset of LCC, where agent nesting is restricted, are very close syntactically and semantically. The first contribution of this paper is to provide translations between CSR and flat-LCC and back. The second contribution is a transformation from the full LCC language to flat-LCC which preserves semantics. This transformation is similar to λ-lifting in functional languages. In conjunction with the equivalence between CHR and CSR with respect to naive operational semantics, these results lead to semantics-preserving translations from full LCC to CHR and conversely. Immediate consequences of this work include new proofs for CHR linear logic and phase semantics, relying on corresponding results for LCC, plus an encoding of the λ-calculus in CHR.

1 Introduction

Constraint Handling Rules (CHR) [1] is a rule-based declarative programming language. Programs are sets of transformation rules on constraint stores. Some constraints are built-ins and can only be accumulated into the store. Other constraints are user-defined and can be added or deleted. Although initial motivations were the definition of constraint solvers and propagators, nowadays applications include typing [2,3], software testing [4], scheduling [5] and so on.

Foundations of the class CC of *Concurrent Constraint* programming languages [6] are very close to CHR: both are based on a model of concurrent computation, where agents communicate through a shared constraint store, with a synchronization mechanism based on constraint entailment. In classical constraint settings, the store evolves monotonically, similarly to the built-in constraint store of CHR. The LCC languages [7,8] introduce linear constraint systems, based on Girard's intuitionistic linear logic (ILL) [9]. A remarkable kind of linear constraints are linear tokens [8], which can be freely added or consumed, comparably to CHR constraints. Linear logic leads to a natural semantics for classical CC languages as well [8]. More recently, a precise declarative semantics for CHR has been described in linear logic [10].

This paper formalizes connections between CHR with naive operational semantics and LCC. Two translations from CHR to LCC and back are proposed, both preserving the semantics. Strong bisimilarity results are formulated. As

direct corollary, we obtain a natural encoding of the λ-calculus in CHR. While existence of low-level translations is guaranteed by Turing-completeness *via* a compilation process, there are more fine-grained criteria to compare expressiveness [11]. In particular, translations presented here are natural and (relatively) agnostic with respect to the constraint theory.

Section 2 presents CHR and LCC in full generality and recalls some already published and well-known results. Section 3 focuses on distinguished subsets *Constraint Simplification Rules* (CSR) and flat-LCC, provides translations from flat-LCC to CHR and back. Linear logic semantics [10] and phase semantics [12] of CHR are recovered as corollary. Section 4 introduces the *ask-lifting* transformation from full LCC to flat-LCC.

Related work

The adaptations of functional concepts in LCC languages have been initiated in Rémy Haemmerlé's PhD thesis [13] with the embedding of closures and modules, leading to an encoding of λ-calculus in LCC. This paper pursues the effort of transposing results in functional languages to concurrent constraint systems.

The translation from full LCC to CHR relies on *ask-lifting*. This is a transformation comparable to the λ-lifting [14] for functional languages: the common idea is the materialization of the environment in data structures, *i.e.* values in functional languages or tokens in LCC.

Flattening nested programming structures to CHR programs was suggested in [15] for connecting the Celf system [16] to CHR but, to our knowledge, no formal description of the transformation has been published.

2 Syntax and Semantics of CHR and LCC

We will denote by $\mathcal{V}$ a set of variables, and by Σ a signature for constant, function and predicate symbols. The set of free variables of a formula e is denoted $\mathrm{fv}(e)$, a sequence of variables is denoted by $\boldsymbol{x}$. $e[\boldsymbol{t}/\boldsymbol{x}]$ denotes the formula e in which free occurrences of variables $\boldsymbol{x}$ are substituted by terms $\boldsymbol{t}$ (with the usual renaming of bound variables to avoid variable clashes).

For a set S, $S^\star$ denotes the set of finite sequences of elements of S and $\mathcal{M}(S)$ denotes the set of finite multi-sets of elements of S. More formally, $(S^\star; \cdot; \varepsilon)$ denotes the free monoid and $(\mathcal{M}(S); , ; \varnothing)$ the free commutative monoid over the elements of S. For relations $\mathcal{R}$ and $\mathcal{R}'$, $a\,\mathcal{R}\cdot\mathcal{R}'\,c$ if there exists b such that $a\,\mathcal{R}\,b\,\mathcal{R}'\,c$. For a relation $\rightarrow$, $\xrightarrow{\star}$ is the reflexive and transitive closure of $\rightarrow$.

2.1 Syntax and Semantics of CHR

Let $\mathcal{P}_b$ and $\mathcal{P}_c$ be two disjoint subsets of predicate symbols in Σ. Predicates built from Σ with predicate symbols in $\mathcal{P}_b$ are *atomic built-in constraints*, their set is denoted $\mathcal{B}_0$. *Built-in constraints* are conjunctions of atomic built-in constraints, their set is denoted $\mathcal{B}$. Predicates built from Σ with predicate symbols in $\mathcal{P}_c$ are

atomic CHR *constraints*, their set is denoted $\mathcal{U}_0$. CHR *constraints* are (finite) multi-sets of atomic CHR constraints, their set is denoted $\mathcal{U}$. A *goal* is a multi-set of built-in constraints and CHR constraints.

Definition 1 (Syntax). *A* CHR program *is a set of* rules*, each rule being denoted* $\langle H \backslash H' \Leftrightarrow G \mid B \rangle$ *where* heads H *and* H' *are* CHR *constraints such that* $\langle H, H' \rangle \neq \varnothing$*, the* guard G *is a built-in constraint, and the* body B *is a goal.*

Example 1. The CHR program below, adapted from [17], describes the *dining philosophers* protocol [18], where N philosophers are sitting around a table and alternate thinking and eating. N forks are dispatched between them. Each philosopher is in competition with her neighbors to take her two adjacent forks and eat.

```
diner(N) ⇔ recphilo(0, N).
recphilo(I, N) ⇔
    J is (I + 1) mod N, philo(I, J), fork(I), nextphilo(I, N).
nextphilo(I, N) ⇔ I < N − 1 | J is I + 1, recphilo(J, N).
philo(I, J) \ fork(I), fork(J) ⇔ eat(I, J).
eat(I, J) ⇔ fork(I), fork(J).
```

Built-in constraints are supposed to include the syntactic equality $=$. There is a *constraint theory* CT over the built-in constraints: CT is supposed to be a non-empty, consistent and decidable first-order theory. For two multi-sets $H = (H_1, \ldots, H_m)$ and $H' = (H'_1, \ldots, H'_n)$, $H \doteq H'$ denotes the formula $H_1 = H'_1 \wedge \cdots \wedge H_n = H'_n$ if $m = n$, and false if $m \neq n$ [19].

A *state* is a tuple denoted $\langle g; b; c \rangle_V$ where g is a goal, b is a built-in constraint, c is a CHR constraint and V is a set of variables. The relation $\equiv_C$ over states is the smallest equivalence relation such that:

- $\langle g; b; c \rangle_V \equiv_C \langle g; b'; c \rangle_V$ for $CT \models b \leftrightarrow b'$;
- $\langle g; b; c \rangle_V \equiv_C \langle g; b; c \rangle_V [y/x]$ for variables $x \notin V$ and $y \notin V \cup \mathrm{fv}(g, b, c)$.

Let $\mathcal{P}$ be the set of pairs of CHR programs and states.

Definition 2 (Naive Operational Semantics [1]). *A* CHR *program* P *is executed along a* transition relation $\rightarrow_P$ *over states:*

Firing Rule

Apply

$$\frac{\langle H \backslash H' \Leftrightarrow G \mid B \rangle \text{ is a fresh variant of a rule in } P \text{ with variables } \boldsymbol{x} \qquad CT \models \forall (b \rightarrow \exists \boldsymbol{x} (H \doteq h \wedge H' \doteq h' \wedge G))}{\langle g; b; h, h', c \rangle_V \rightarrow_P \langle B, g; H \doteq h \wedge H' \doteq h' \wedge G \wedge b; h, c \rangle_V}$$

Solving Rules

Solve

$$\frac{B \in \mathcal{B} \qquad CT \models B \wedge b \leftrightarrow b'}{\langle B, g; b; c \rangle_V \rightarrow_P \langle g; b'; c \rangle_V}$$

Introduce

$$\frac{C \in \mathcal{U}}{\langle C, g; b; c \rangle_V \rightarrow_P \langle g; b; C, c \rangle_V}$$

Let q *be an initial goal, the* query*.* V *is defined as* $\mathrm{fv}(q)$ *and, from the* initial state $s_0 = \langle q; \top; \varnothing \rangle_V$*, a* derivation *is a sequence* $s_0 \rightarrow_P s_1 \rightarrow_P \cdots \rightarrow_P s_n$*. Such a state* s_n *is an* accessible state.

Definition 3 (Linear Logic Semantics [10]).
For built-in constraint $B = \langle B_1 \wedge \cdots \wedge B_n \rangle$, *let* $B^\dagger = \langle !B_1 \otimes \cdots \otimes !B_n \rangle$.
For CHR *constraint* $C = (C_1, \ldots, C_n)$, *let* $C^\dagger = \langle C_1 \otimes \cdots \otimes C_n \rangle$.
For goal $G = (G_1, \ldots, G_n)$, *let* $G^\dagger = \langle G_1^\dagger \otimes \cdots \otimes G_n^\dagger \rangle$.
For state $S = \langle g; b; c \rangle_V$, *let* $S^\dagger = \exists \boldsymbol{x}(g^\dagger \otimes b^\dagger \otimes c^\dagger)$, *where* $\boldsymbol{x} = \mathrm{fv}(G, B, C) \setminus V$.
The semantics of a rule $r = \langle H \backslash H' \Leftrightarrow G \mid B \rangle$ *is, with* $\boldsymbol{x} = \mathrm{fv}(B) \setminus \mathrm{fv}(H, H', G)$: $r^\dagger = \langle !\forall (G^\dagger \otimes H^\dagger \otimes H'^\dagger \multimap \exists \boldsymbol{x}(H^\dagger \otimes B^\dagger)) \rangle$.
The linear logic semantics of a program $P = \{r_1, \ldots, r_n\}$ *is* $P^\dagger = \langle r_1^\dagger \otimes \cdots \otimes r_n^\dagger \rangle$.

Theorem 1 (Soundness & Completeness [10]).
Let $CT^\dagger$ *be the Girard translation of* CT *[9],* P *a* CHR *program and* q *a query.*

- *(Sound) If* s *is an accessible state from* q *in* P, *then* $P^\dagger, CT^\dagger \models \forall (q^\dagger \multimap s^\dagger)$.
- *(Complete) For every formula* c *such that* $P^\dagger, CT^\dagger \models \forall (q^\dagger \multimap c)$, *there is an accessible state* s *from* q *in* P *such that* $CT^\dagger \models \forall (s^\dagger \multimap c)$.

2.2 Syntax and Semantics of LCC

Definition 4 (Linear Constraint System [8]). *A* linear constraint system *is a pair* $(\mathcal{C}, \vdash_\mathcal{C})$, *where:*

- $\mathcal{C}$ *is a set of formulas (the* linear constraints*) built from variables* $\mathcal{V}$ *and the signature* Σ, *with logical operators: multiplicative conjunction* $\otimes$, *its neutral* 1, *existential* $\exists$, *exponential* $!$ *and constant* $\top$; $\mathcal{C}$ *is assumed to be closed by renaming, multiplicative conjunction and existential quantification;*
- $\Vdash_\mathcal{C}$ *is a binary relation over* $\mathcal{C}$, *which defines the non-logical axioms.*
- $\vdash_\mathcal{C}$ *is the least subset of* $\mathcal{C}^\star \times \mathcal{C}$ *containing* $\Vdash_\mathcal{C}$ *and closed by the rules of intuitionistic multiplicative exponential linear logic for* 1, $\top$, $\otimes$, $!$ *and* $\exists$.

Definition 5 (Syntax with Persistent Asks [20]). *The syntax for building* LCC agents *follows the grammar:* $A ::= \forall \mathcal{V}^\star(\mathcal{C} \to A) \,|\, \forall \mathcal{V}^\star(\mathcal{C} \Rightarrow A) \,|\, \exists \mathcal{V}.A \,|\, \mathcal{C} \,|\, A \| A$ *where* $\|$ *stands for parallel composition,* $\exists$ *for variable hiding,* $\to$ *for (transient) ask and* $\Rightarrow$ *for persistent ask. In the particular case where there are no universally quantified variables in an ask, the notation* $(c \to a)$ *is preferred to* $\forall \varepsilon (c \to a)$.

Agent $\forall \boldsymbol{x}(c \to a)$ suspends until c is entailed then wakes up and does a. Transient asks wake up at most one time. Persistent asks are introduced [20] to replace declarations by agents. The agent $\forall \boldsymbol{x}(c \Rightarrow a)$ can wake up as many times as c is entailed. This behavior makes sense as entailment consumes resources.

Example 2. Here is the LCC version for dining philosophers [8,13].

$$
\begin{array}{l}
\forall N(\,\mathrm{diner}(N) \Rightarrow \\
\quad \exists K(\forall I(\,\mathrm{recphilo}(K, I) \Rightarrow \\
\qquad \mathrm{fork}(K, I) \,\| \\
\qquad \exists J.(\,J \textbf{ is } (I + 1) \bmod N \,\| \\
\qquad\quad (\,\mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J) \Rightarrow \\
\qquad\quad\ \ \mathrm{eat}(K, I) \,\| \, (\mathrm{eat}(K, I) \to \mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J)) \,\| \\
\qquad\quad (I < N - 1 \to \mathrm{recphilo}(K, J)) \,\| \\
\qquad \mathrm{recphilo(K, 0)})))))
\end{array}
$$

This example makes usage of non-trivial scopes for variables: N, K, I and J are in turn introduced and shared by subsequent asks. The recursive loop $\langle$recphilo$\rangle$ installs N forks and composes N agents (the philosophers) in parallel. The variable K identifies tokens and let to run several $\langle$diner$\rangle$ in parallel (a *banquet* [13]) while preventing tables from stealing cutlery from each other. The philosopher between forks I and J is an agent in LCC, whereas she is materialized in example 1 by the CHR constraint philo(I, J) in order to carry the environment $\{\mathrm{I}, \mathrm{J}\}$.

A *configuration* is a triple $(X; c; \Gamma)$ where c is a constraint (the *store*), Γ is a multi-set of agents and X is a set of variables (the *hidden variables*). The relation $\equiv_{\mathrm{L}}$ over configurations is the smallest equivalence relation such that:

- $(X; c; a \parallel b, \Gamma) \equiv_{\mathrm{L}} (X; c; a, b, \Gamma)$ for all agents a and b;
- $(X; c; 1, \Gamma) \equiv_{\mathrm{L}} (X; c; \Gamma)$;
- $(X; c; \Gamma) \equiv_{\mathrm{L}} (X; c'; \Gamma)$ for all constraints c, c' such that $c \dashv\vdash_{\mathcal{C}} c'$;
- $(X; c; \Gamma) \equiv_{\mathrm{L}} (X; c; \Gamma)[y/x]$ for all variables $x \in X$ and $y \notin \mathrm{fv}(X, c, \Gamma)$

Let $\mathcal{K}$ be the set of configurations.

Definition 6 (Operational Semantics [8,20]). *The* transition relation $\rightarrow_{\mathrm{L}}$ *is the least relation on configurations satisfying the following rules:*

Firing Rules

Transient Ask

$$\frac{c \vdash_{\mathcal{C}} \exists Y (d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d' ((c \vdash_{\mathcal{C}} \exists Y (d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash_{\mathcal{C}} d')}{(X; c; \forall \boldsymbol{x} (e \rightarrow a), \Gamma) \rightarrow_{\mathrm{L}} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \Gamma)}$$

Persistent Ask

$$\frac{c \vdash_{\mathcal{C}} \exists Y (d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d' ((c \vdash_{\mathcal{C}} \exists Y (d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash_{\mathcal{C}} d')}{(X; c; \forall \boldsymbol{x} (e \Rightarrow a), \Gamma) \rightarrow_{\mathrm{L}} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \forall \boldsymbol{x} (e \Rightarrow a), \Gamma)}$$

Solving Rules

Hiding

$$\frac{y \notin X \cup \mathrm{fv}(c, \Gamma)}{(X; c; \exists x.a, \Gamma) \rightarrow_{\mathrm{L}} (X \cup \{y\}; c \otimes d; a[y/x], \Gamma)}$$

Tell

$$\frac{}{(X; c; d, \Gamma) \rightarrow_{\mathrm{L}} (X; c \otimes d; \Gamma)}$$

Equivalence

$$\frac{\kappa_0 \equiv_{\mathrm{L}} \kappa_0' \rightarrow_{\mathrm{L}} \kappa_1' \equiv_{\mathrm{L}} \kappa_1}{\kappa_0 \rightarrow_{\mathrm{L}} \kappa_1}$$

An agent a is associated with the initial configuration $(\varnothing; \top; a)$. Accessible observables *from a configuration κ are the configurations κ' such that $\kappa \xrightarrow{\star}_{\mathrm{L}} \kappa'$.*

Definition 7 (Linear Logic Semantics [8,20]). *The translation $(\cdot)^{\ddagger}$ of* LCC *agents into their* linear logic semantics *is defined inductively as follows:*

$$(\forall \boldsymbol{x} (c \rightarrow a))^{\ddagger} = \forall \boldsymbol{x} (c \multimap a^{\ddagger}) \qquad (\forall \boldsymbol{x} (c \Rightarrow a))^{\ddagger} = !\forall \boldsymbol{x} (c \multimap a^{\ddagger})$$

$$(\exists x.a)^{\ddagger} = \exists x (a^{\ddagger}) \qquad c^{\ddagger} = c \qquad (a \parallel b)^{\ddagger} = a^{\ddagger} \otimes b^{\ddagger}$$

If Γ is a multi-set of agents $(a_1, \ldots, a_n)$, we define $\Gamma^\ddagger = \langle a_1^\ddagger \otimes \cdots \otimes a_n^\ddagger \rangle$. Configurations are translated to $(X; c; \Gamma)^\ddagger = \langle \exists X (c \otimes \Gamma^\ddagger) \rangle$.

Theorem 2 (Soundness & Completeness [8,20,13]). *For all agents a:*

- *(Sound) If κ is an accessible observable from $(\varnothing; \top; a)$, then $a^\ddagger \vdash_{\mathcal{C}} \kappa^\ddagger$.*
- *(Complete) If c is such that $a^\ddagger \vdash_{\mathcal{C}} c$, then there is an accessible observable $(X; d; \Gamma)$ from $(\varnothing; \top; a)$ with $\exists X (d) \vdash_{\mathcal{C}} c$ and agents in Γ are persistent asks.*

2.3 Circumscribing non-determinism in CHR and LCC operational semantics

Whereas non-determinism in firing rules seems to be inherent to the computation model (and is tackled in CHR by the committed-choice strategy and by the refined semantics), the non-determinism in sequencing solving rules can be completely eliminated. This is a classical result for constraint logic programming [21] and it was proved for LCC in [13]. We formalize such a result for CHR and LCC since the precise bisimulation results presented in next sections rely on it.

Let $\rightarrow_P^s$ and $\rightarrow_P^f$ be the restrictions of $\rightarrow_P$ to solving and firing rules respectively. Let $\rightarrow_{\mathrm{L}}^s$ and $\rightarrow_{\mathrm{L}}^f$ be the similar restrictions for $\rightarrow_{\mathrm{L}}$.

We define $\Rightarrow_P^s$ such that $s \Rightarrow_P^s s'$ if and only if $s \xrightarrow{\star}_P^s s' \not\rightarrow_P^s$. Similarly, $\Rightarrow_{\mathrm{L}}^s$ is such that $\kappa \Rightarrow_{\mathrm{L}}^s \kappa'$ if and only if $\kappa \xrightarrow{\star}_{\mathrm{L}}^s \kappa' \not\rightarrow_{\mathrm{L}}^s$.

Lemma 1 (Solving rules terminate and are confluent modulo $\equiv$).
For every CHR *program P, for all state s, there exists s' such that $s \Rightarrow_P^s s'$ and for all s', s'', if $s \Rightarrow_P^s s'$ and $s \Rightarrow_P^s s''$, then $s' \equiv_{\mathrm{C}} s''$.*
For every configuration κ, there exists κ' such that $\kappa \Rightarrow_{\mathrm{L}}^s \kappa'$ and for all κ', κ'', if $\kappa \Rightarrow_{\mathrm{L}}^s \kappa'$ and $\kappa \Rightarrow_{\mathrm{L}}^s \kappa''$ then $\kappa' \equiv_{\mathrm{L}} \kappa''$.

Thus, observed configurations can be restricted to be final for $\rightarrow^s$ (or, equivalently, normalized by $\Rightarrow^s$) without losing derivations. The following lemma is a specialization of the "*Andorra*" principle [22] to the rule selection strategy:

Lemma 2 (*Full solving before firing*). *For every* CHR *program P,*

$$\left(\xrightarrow{\star}_P \cdot \Rightarrow_P^s\right) = \left((\Rightarrow_P^s \cdot \rightarrow_P^f)^\star \cdot \Rightarrow_P^s\right)$$

and, similarly,

$$\left(\xrightarrow{\star}_{\mathrm{L}} \cdot \Rightarrow_{\mathrm{L}}^s\right) = \left((\Rightarrow_{\mathrm{L}}^s \cdot \rightarrow_{\mathrm{L}}^f)^\star \cdot \Rightarrow_{\mathrm{L}}^s\right)$$

The lemma 2 is a corollary of the monotonous selection strategy [13]: intuitively, $\rightarrow^s$ can always be exhausted before applying $\rightarrow^f$.

Lemma 3 (Solving rules preserve declarative semantics). *For every* CHR *program P, if $s \rightarrow_P^s s'$, then $s^\dagger \equiv s'^\dagger$. Similarly, if $\kappa \rightarrow_{\mathrm{L}}^s \kappa'$, then $\kappa^\ddagger \equiv \kappa'^\ddagger$.*

Therefore, next sections focus on $\Rightarrow$-transitions where $\Rightarrow_P = (\Rightarrow_P^s \cdot \rightarrow_P^f \cdot \Rightarrow_P^s)$, and $\Rightarrow_{\mathrm{L}} = (\Rightarrow_{\mathrm{L}}^s \cdot \rightarrow_{\mathrm{L}}^f \cdot \Rightarrow_{\mathrm{L}}^s)$: a $\Rightarrow_P$*-accessible state* from s is a state s' such that $s \stackrel{\star}{\Rightarrow}_P s'$ and a $\Rightarrow_{\mathrm{L}}$*-accessible observable* from κ is a configuration κ' such that $\kappa \stackrel{\star}{\Rightarrow}_{\mathrm{L}} \kappa'$. It is worth noticing that a firing occurs at each $\Rightarrow$-transition.

3 Translations between sub-languages CSR and flat-LCC

From now on, we consider the linear constraint system $(\mathcal{C}, \vdash_{\mathcal{C}})$ induced by the constraint theory CT and with atomic CHR constraints as linear tokens. More precisely, $\mathcal{C}$ is the least set of formulas which contains $\top$ and $!B$ for all $B \in \mathcal{B}_0$ and C for all $C \in \mathcal{U}_0$, closed by renaming, multiplicative conjunction and existential quantification. We suppose that $c \Vdash_{\mathcal{C}} d$ if and only if $CT^{\dagger} \models \forall(c \multimap d)$. The result is a particular form of linear constraint system where non-logical axioms follow from the translation of a classical theory.

Bisimulation is the most popular method for comparing concurrent processes [23], characterizing a notion of strong equivalence between processes. A *transition system* is a tuple $(S, \rightarrow)$ with S a set of states and $\rightarrow$ a binary relation over S. We define the CHR transition system as $(\mathcal{P}, \Rightarrow_{\mathrm{C}})$ where $(P, s) \Rightarrow_{\mathrm{C}} (P', s')$ when $P = P'$ and $s \Rightarrow_P s'$, and the LCC transition system as $(\mathcal{K}, \Rightarrow_{\mathrm{L}})$.

Definition 8 (Bisimulation). *Let* $(S_1, \xrightarrow{1})$ *and* $(S_2, \xrightarrow{2})$ *be two transition systems. A* bisimulation *is a relation* $\sim \subseteq S_1 \times S_2$ *such that for all* $s_1 \sim s_2$*:*

- *for all* s_1' *such that* $s_1 \xrightarrow{1} s_1'$*, there exists* s_2' *such that* $s_2 \xrightarrow{2} s_2'$ *and* $s_1' \sim s_2'$*;*
- *for all* s_2' *such that* $s_2 \xrightarrow{2} s_2'$*, there exists* s_1' *such that* $s_1 \xrightarrow{1} s_1'$ *and* $s_1' \sim s_2'$*.*

3.1 From Constraint Simplification Rules (CSR) to flat-LCC

Resulting configurations of LCC Firing Rules enjoy a new store where guards have been consumed. This behavior corresponds to simplification rules in CHR.

Definition 9 (CSR programs[19]). *A* CHR *program* P *is a* CSR program *when all rules of* P *are simplifications (*i.e. *rules are of the form* $\langle H \Leftrightarrow G \mid B. \rangle$*).*

As far as naive operational semantics and linear-logic semantics are concerned, expressiveness of CHR and CSR is identical. For a rule $r = \langle H \backslash H' \Leftrightarrow G \mid B \rangle$, let $r^{\times} = \langle H, H' \Leftrightarrow G \mid H, B. \rangle$ and for $P = \{r_1, \ldots, r_n\}$, let $P^{\times} = \{r_1^{\times}, \ldots, r_n^{\times}\}$.

Example 3. Here is leq$^{\times}$ translated from a version of the leq program [24]:

leq(X, X) $\Leftrightarrow$ true.
leq(X, Y) $\Leftrightarrow$ number(X), number(Y) | X $\leqslant$Y.
leq(X, Y), leq(Y, X) $\Leftrightarrow$ X = Y.
leq(X, Y), leq(Y, Z) $\Leftrightarrow$ leq(X, Y), leq(Y, Z), leq(X, Z).
leq(X, Y), leq(X, Y) $\Leftrightarrow$ leq(X, Y).

Proposition 1 (CHR and CSR equivalence). *For every* CHR *program* P*, we have* $\rightarrow_P = \rightarrow_{P^{\times}}$ *and* $P^{\dagger} \equiv (P^{\times})^{\dagger}$*.*

This equivalence only holds for naive CHR semantics. There is probably no natural encoding of the traditional semantics for propagation [25] in LCC, at least without *ad-hoc* support hard-wired in the constraint system.

Let $r = \langle H' \Leftrightarrow G \,|\, B. \rangle$ be a simplification rule. $G^\dagger \otimes H'^\dagger$ and $B^\dagger$ are in $\mathcal{C}$, thus the following agent is well-formed: $r^{-\circ} = \langle \forall \boldsymbol{y}(G^\dagger \otimes H'^\dagger \Rightarrow \exists \boldsymbol{x}.B^\dagger) \rangle$, where $\boldsymbol{x} = \mathrm{fv}(B) \setminus \mathrm{fv}(H', G)$ and $\boldsymbol{y} = \mathrm{fv}(H', G)$. For every CSR program $P = \{r_1, \ldots, r_n\}$, the *translation of* P *in* LCC is: $P^{-\circ} = \langle r_1^{-\circ} \parallel \ldots \parallel r_n^{-\circ} \rangle$. States $\langle g; b; c \rangle_V$ are translated in $\mathcal{C}$ as well: $\langle g; b; c \rangle_V{}^{-\circ} = g^\dagger \otimes b^\dagger \otimes c^\dagger$.

Example 4. The leq$^\times$ program (Example 3) is translated to the agent leq$^{-\circ}$:

$$\begin{aligned}\mathrm{leq}^{-\circ} = {} & \forall X(\mathrm{leq}(X,X) \Rightarrow 1) \parallel \\ & \forall XY(\mathrm{number}(X) \otimes \mathrm{number}(Y) \otimes \mathrm{leq}(X,Y) \Rightarrow X \leqslant Y) \parallel \\ & \forall XY(\mathrm{leq}(X,Y) \otimes \mathrm{leq}(Y,X) \Rightarrow X = Y) \parallel \\ & \forall XYZ(\mathrm{leq}(X,Y) \otimes \mathrm{leq}(Y,Z) \Rightarrow \mathrm{leq}(X,Y) \otimes \mathrm{leq}(Y,Z) \otimes \mathrm{leq}(X,Z)) \parallel \\ & \forall XY(\mathrm{leq}(X,Y) \otimes \mathrm{leq}(X,Y) \Rightarrow \mathrm{leq}(X,Y))\end{aligned}$$

Since there is no possible confusion between linear tokens and classical constraints, then, by abuse of notations, we omit the ! operator on $\mathcal{U}_0$ constraints.

Definition 10 (CSR **to** LCC **translation).** *A* CSR *program P and a query q are translated to the agent $a(P,q) = \langle P^{-\circ} \| q^\dagger \rangle$.*

Main Result 1 (Bisimilarity) *Let $\sim \subseteq \mathcal{P} \times \mathcal{K}$ be the relation where $(P,s) \sim \kappa$ if and only if $\kappa \equiv_{\mathrm{L}} (X; s^{-\circ}; P^{-\circ})$ with $X = \mathrm{fv}(s) \setminus V$. Then, $\sim$ is a bisimulation.*

Corollary 1 (Semantics preservation). *For* CSR *program P, query q:*

- *if κ is a $\Rightarrow_{\mathrm{L}}$-accessible observable of $a(P,q)$, then $\kappa \equiv (X; c; P^{-\circ})$ and there is a $\Rightarrow_P$-accessible state s from q with $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$, $\boldsymbol{x} = \mathrm{fv}(s) \setminus \mathrm{fv}(q)$;*
- *if s is a $\Rightarrow_P$-accessible state from q, then there is a $\Rightarrow_{\mathrm{L}}$-accessible observable $(X; c; P^{-\circ})$ from $a(P,q)$ such that $\exists \boldsymbol{x}(s^{-\circ}) \dashv\vdash_{\mathcal{C}} \exists X(c)$, where $\boldsymbol{x} = \mathrm{fv}(s) \setminus \mathrm{fv}(q)$.*

3.2 From flat-LCC to CSR

The translation of CSR into LCC generates agents of the particular form $p \parallel q$, where the sub-agent p is the translation of a CSR program and is therefore a parallel composition of persistent asks without any nested asks, and the sub-agent q is a translation of a query and is therefore reduced to a constraint. Moreover, every ask guard consumes at least a linear token (since CHR heads are non-empty) and asks are closed term (*i.e.* without free variables). Such agents are characterized by the following definition:

Definition 11 (flat-LCC**).** Flat-LCC agents *are restricted to the grammar: $A^\uparrow ::= A^\forall \parallel \mathcal{C}$ where $A^\forall ::= \forall \mathcal{V}^\star(\mathcal{C} \Rightarrow \mathcal{C}) \mid A^\forall \parallel A^\forall \mid 1$ with the following side condition for every ask $\forall \boldsymbol{x}(g \Rightarrow c)$: $g \not\vdash_{\mathcal{C}} g \otimes g$ (consumption) and $\mathrm{fv}(g,c) \subseteq \boldsymbol{x}$.*

This subsection is dedicated to establishing the reverse translation, from $A^\uparrow$ to CSR. It is worth noticing first that, like a CSR program, an $A^\uparrow$-agent essentially transforms constraint stores without introducing new suspensions:

Lemma 4 (Configurations form). *Non-initial $\Rightarrow_{\mathrm{L}}$-accessible configurations from an $A^\uparrow$-agent a are $\equiv_{\mathrm{L}}$-equivalent to configurations of the form $(_; _; a^\forall)$.*

The translation from flat-LCC to CSR is a bit more intricate than the other direction on account of, firstly, splitting between built-in constraints and CHR constraints, and secondly, possible introductions of local variables by $\exists$. Fresh variables should be introduced to translate constraints such as $a(X,Y) \otimes \exists X(b(X,Y))$ into $\langle a(X,Y), b(K,Y)\rangle$ where K is a new local variable. The function $f^{\mathcal{C}}$ translates every constraint in $\mathcal{C}$ to a tuple $(X; B; C)$ where B is a built-in constraint, C a CHR constraint and X a set of variables local to B and C:

$$
\begin{aligned}
f^{\mathcal{C}}(\top) &= (\varnothing; \mathrm{true}; \varnothing) \\
f^{\mathcal{C}}(!B) &= (\varnothing; B; \varnothing) && \text{for all } B \in \mathcal{B}_0 \\
f^{\mathcal{C}}(C) &= (\varnothing; \mathrm{true}; C) && \text{for all } C \in \mathcal{U}_0 \\
f^{\mathcal{C}}(c \otimes d) &= (\sigma_c(X_c) \cup \sigma_d(X_d); \sigma_c(B_c) \wedge \sigma_d(B_d); \sigma_c(C_c), \sigma_d(C_d)) \\
&\quad \text{if } f^{\mathcal{C}}(c) = (X_c; B_c; C_c) \text{ and } f^{\mathcal{C}}(d) = (X_d; B_d; C_d) \\
&\quad \text{with } \sigma_c \text{ and } \sigma_d \text{ renaming of } X_c \text{ and } X_d \text{ respectively} \\
&\quad \text{such that } \sigma_c(X_c) \cap \mathrm{fv}(\sigma_d(B_d, C_d)) = \sigma_d(X_d) \cap \mathrm{fv}(\sigma_c(B_c, C_c)) = \varnothing \\
f^{\mathcal{C}}(\exists x(c)) &= (X_c \cup \{x\}; B_c; C_c) \quad \text{if } f^{\mathcal{C}}(c) = (X_c; B_c; C_c)
\end{aligned}
$$

$A^{\forall}$-agents are translated to CSR programs through the function $f^{\forall}$. Translation of asks should take care of clashes with similar renaming as for $\otimes$ in $f^{\mathcal{C}}$:

$$
\begin{aligned}
f^{\forall}(\forall \boldsymbol{x}(g \Rightarrow c)) &= \{\langle \sigma_g(C_g) \Leftrightarrow \sigma_g(B_g) \mid \sigma_c(B_c), \sigma_c(C_c).\rangle\} \\
&\quad \text{where } f^{\mathcal{C}}(g) = (X_g; B_g; C_g) \text{ and } f^{\mathcal{C}}(c) = (X_c; B_c; C_c) \\
&\quad \text{and } \sigma_g \text{ and } \sigma_c \text{ renaming of } X_g \text{ and } X_c \text{ respectively} \\
&\quad \text{with } \sigma_g(X_g) \cap \mathrm{fv}(\sigma_c(B_c, C_c)) = \sigma_c(X_c) \cap \mathrm{fv}(\sigma_g(B_g, C_g)) = \varnothing \\
f^{\forall}(a \parallel b) &= f^{\forall}(a) \cup f^{\forall}(b) \\
f^{\forall}(1) &= \varnothing
\end{aligned}
$$

For every ask $\forall \boldsymbol{x}(g \Rightarrow c)$, $f^{\forall}(\forall \boldsymbol{x}(g \Rightarrow c))$ is a well-formed CHR rule. In particular, the side condition on g ensures that $\sigma_g(C_g) \neq \varnothing$.

$f^s_V : c \mapsto \langle \varnothing; b; c\rangle_V$ maps constraints to states with $(_; b; c) = f^{\mathcal{C}}(c)$.

Note that all variables in CSR queries are global. The CHR program initialization should hide existentially quantified variables in the top-level constraint c_0 of the agent. We suppose a fresh symbol $\langle \mathrm{start}/n\rangle \in \mathcal{U}_0$ where $n = \#\mathrm{fv}(c_0)$.

Definition 12 (Flat-LCC to CSR translation). *A flat-*LCC *agent* $\langle a^{\forall} \parallel c_0\rangle$ *is translated to the* CHR *program* $P(a^{\forall} \parallel c_0) = f^{\forall}(a^{\forall}) \cup \{\mathrm{start}(\boldsymbol{v}) \Leftrightarrow B_0, C_0.\}$ *and the query* $q(a) = (\mathrm{start}(\boldsymbol{v}))$ *where* $(_; B_0; C_0) = f^{\mathcal{C}}(c_0)$ *and* $\boldsymbol{v} = \mathrm{fv}(c_0)$.

Main Result 2 (Bisimilarity) *Let* $\sim \subseteq \mathcal{K} \times \mathcal{P}$ *be the relation where* $\kappa \sim (P, s)$ *if and only if there exists a flat-*LCC *agent* $\langle a^{\forall} \parallel c_0\rangle$ *where* $\kappa \equiv_{\mathrm{L}} (X; c; a^{\forall})$ *and* $P = P(a)$ *and* $s \equiv_{\mathrm{C}} f^s_V(c)$, *with* $V = \mathrm{fv}(c_0)$. *Then,* $\sim$ *is a bisimulation.*

Corollary 2 (Semantics preservation). *For every flat-*LCC *agent* $a = \langle a^{\forall} \parallel c_0\rangle$, *let* $s_0 = \langle q(a); \top; \varnothing\rangle_V$, $V = \mathrm{fv}(c_0)$, *then:*

- *for all* $\Rightarrow_{\mathrm{L}}$*-accessible configuration* $(X; c; a^{\forall})$ *from* a, *there exists* $a \Rightarrow_{P(a)}$*-accessible state* s *from* s_0 *such that* $\exists \boldsymbol{x}(s^{\multimap}) \dashv\vdash_{\mathcal{C}} \exists X(c)$;
- *for all* $\Rightarrow_{P(a)}$*-accessible state* s *from* s_0, *if* $s \neq s0$, *there exists* $a \Rightarrow_{\mathrm{L}}$*-accessible configuration* $(X; c; a^{\forall})$ *from* a, *such that* $\exists \boldsymbol{x}(s^{\multimap}) \dashv\vdash_{\mathcal{C}} \exists X(c)$;

where, in both cases, $\boldsymbol{x} = \mathrm{fv}(s) \setminus V$.

3.3 CHR linear-logic and phase semantics revisited

Lemma 5 (Identical Semantics). *For every* CSR *program* P *and query* q, $(P^{-\circ})^{\ddagger} \equiv P^{\dagger}$ *and we have* $P^{\dagger}, CT^{\dagger} \models \forall(q^{\dagger} \multimap c)$ *if and only if* $(P^{-\circ})^{\ddagger}, q^{\dagger} \vdash_{\mathcal{C}} c$.

Relating theorem 1 and theorem 2 supposes to prove that accessible constraints are included in provable constraints (correctness), and conversely (completeness). Thus, the correctness and completeness result amounts to equality between sets, which we make explicit here to prove both ways at the same time.

$$\mathcal{O}_{\mathrm{C}}(P,q) = \left\{(P,s) \in \mathcal{P} \;\middle|\; (q;\top;\varnothing) \xrightarrow{\star}_{P} s\right\} \qquad \mathcal{O}_{\mathrm{L}}(a) = \left\{\kappa \in \mathcal{K} \;\middle|\; (\varnothing;\top;a) \xrightarrow{\star}_{\mathrm{L}} \kappa\right\}$$

$$\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P,q) = \left\{(P,s) \in \mathcal{P} \;\middle|\; (q;\top;\varnothing) \overset{\star}{\Rightarrow}_{P} s\right\} \qquad \mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a) = \left\{\kappa \in \mathcal{K} \;\middle|\; (\varnothing;\top;a) \overset{\star}{\Rightarrow}_{\mathrm{L}} \kappa\right\}$$

$$\mathcal{LL}_{\mathrm{C}}(P,q) = \left\{c \in \mathcal{C} \;\middle|\; P^{\dagger}, CT^{\dagger} \models \forall(q^{\dagger} \multimap c)\right\} \qquad \Downarrow^{s}_{\mathrm{C}} S = \left\{s \in \mathcal{P} \;\middle|\; \exists s' \in S \Rightarrow^{s}_{\mathrm{C}} s\right\}$$

$$\mathcal{LL}_{\mathrm{L}}(a) = \left\{c \in \mathcal{C} \;\middle|\; a^{\ddagger} \vdash_{\mathcal{C}} c\right\} \qquad \Downarrow^{s}_{\mathrm{L}} S = \left\{\kappa \in \mathcal{K} \;\middle|\; \exists \kappa' \in S \Rightarrow^{s}_{\mathrm{L}} \kappa\right\}$$

$$\downarrow\!\!\circ\, S = \left\{c \in \mathcal{C} \;\middle|\; \exists c' \in S, c' \vdash_{\mathcal{C}} c\right\}$$

Some results mentioned up to now are summarized in the following table:

For every CSR program P, query q, and flat-LCC agent a,

- Theorem 1: $\downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P,q))^{\dagger} = \mathcal{LL}_{\mathrm{C}}(P,q)$;
- Theorem 2: $\downarrow\!\!\circ(\mathcal{O}_{\mathrm{L}}(a))^{\ddagger} = \mathcal{LL}_{\mathrm{L}}(a)$;
- Lemma 2: $\Downarrow^{s}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P,q) = \mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P,q)$ and $\Downarrow^{s}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a) = \mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a)$;
- Lemma 3: $(\Downarrow^{s}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P,q))^{\dagger} = (\mathcal{O}_{\mathrm{C}}(P,q))^{\dagger}$ and $(\Downarrow^{s}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a))^{\ddagger} = (\mathcal{O}_{\mathrm{L}}(a))^{\ddagger}$;
- Proposition 1: $\mathcal{O}_{\mathrm{C}}(P,q) = \mathcal{O}_{\mathrm{C}}(P^{\times},q)$ and $\mathcal{LL}_{\mathrm{C}}(P,q) = \mathcal{LL}_{\mathrm{C}}(P^{\times},q)$;
- Corollary 1: $(\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P^{\times},q))^{\dagger} = (\mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger}$;
- Lemma 5: $\mathcal{LL}_{\mathrm{L}}(P^{\times},q) = \mathcal{LL}_{\mathrm{L}}(a(P^{\times},q))$

We are now ready to prove theorem 1 again from the other results.

Proof of theorem 1. For every CHR program P and query q:

$$\begin{aligned}
\downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P,q))^{\dagger} &\overset{\text{proposition 1}}{=} \downarrow\!\!\circ(\mathcal{O}_{\mathrm{C}}(P^{\times},q))^{\dagger} && \overset{\text{lemma 3}}{=} \downarrow\!\!\circ(\Downarrow^{s}_{\mathrm{C}}\mathcal{O}_{\mathrm{C}}(P^{\times},q))^{\dagger} \\
&\overset{\text{lemma 2}}{=} \downarrow\!\!\circ(\mathcal{O}^{\Rightarrow}_{\mathrm{C}}(P^{\times},q))^{\dagger} && \overset{\textbf{corollary 1}}{=} \downarrow\!\!\circ(\mathcal{O}^{\Rightarrow}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} \\
&\overset{\text{lemma 2}}{=} \downarrow\!\!\circ(\Downarrow^{s}_{\mathrm{L}}\mathcal{O}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} && \overset{\text{lemma 3}}{=} \downarrow\!\!\circ(\mathcal{O}_{\mathrm{L}}(a(P^{\times},q)))^{\ddagger} \\
&\overset{\textbf{theorem 2}}{=} \mathcal{LL}_{\mathrm{L}}(a(P^{\times},q)) && \overset{\text{lemma 5}}{=} \mathcal{LL}_{\mathrm{C}}(P^{\times},q) \\
&\overset{\text{proposition 1}}{=} \mathcal{LL}_{\mathrm{C}}(P,q) && \qquad\qquad \square
\end{aligned}$$

The following proposition describes a method to prove unreachability property in CHR using phase semantics, adapted from similar result in LCC [8].

Proposition 2 (Safety through Phase Semantics [12]). *To prove a safety property of the kind* $s \not\rightarrow_{P} s'$ *for a given* CHR *program* P, *it is enough to prove that for a well-chosen phase space* $\mathbf{P}$ *and valuation* η *compatible with* CT *and* P, *there exists an element* $a \in \eta(s^{\dagger})$ *such that* $a \notin \eta(t^{\dagger})$.

Such a valuation η is compatible with $\vdash_{\mathcal{C}}$ and $(P^{\times})^{-\circ}$ (note that it is immediate to see $(P^{\times})^{-\circ}$ as declarations in the sense of the original presentation of LCC [8]). Let κ and κ' be the respective images of s and s' by the transformation. Then the element a is such that $a \in \eta(\kappa^{\ddagger})$ and $a \notin \eta(\kappa'^{\ddagger})$. Thus $\kappa \not\rightarrow_{\mathrm{L}} \kappa'$ comes from the phase semantics of LCC. Therefore the property $s \not\rightarrow_{P^{\times}} s'$ follows from corollary 1, and is generalizable to P with proposition 1. That proves proposition 2. $\square$

4 Ask-lifting: encoding LCC into CSR

The main result of this section is a translation from LCC to flat-LCC which preserves the semantics. Consequently, thanks to corollary 2, we can deduce a semantics-preserving translation from LCC to CSR. This section begins with a preliminary step introducing an intermediary language LCC^ℓ where asks are labeled with linear tokens: these tokens do not change the operational semantics and there is a trivial labeling to transform LCC programs to LCC^ℓ programs. These linear tokens are introduced in order to follow asks through the transitions of the operational semantics, which is used to prove the semantics preservation.

4.1 Preliminary step: labeling LCC-agents

Labeled LCC agents A^ℓ differ from agents A by labels inserted on each ask. In the following definition, labels are arbitrary linear tokens.

Definition 13 (LCC^ℓ agents). *The syntax of* LCC^ℓ agents *is given by the following grammar:* $A^\ell ::= \forall \mathcal{V}^\star(\mathcal{C} \xrightarrow{\mathcal{U}_0} A^\ell) \mid \forall \mathcal{V}^\star(\mathcal{C} \overset{\mathcal{U}_0}{\Longrightarrow} A^\ell) \mid \exists \mathcal{V}.A^\ell \mid \mathcal{C} \mid A^\ell \parallel A^\ell$.

The transition relation $\rightarrow_{\mathrm{L}}$ is lifted to the transition $\rightarrow_{\mathrm{LCC}^\ell}$ for LCC^ℓ.

$$\textsc{Transient Ask (with labeling)}\quad \frac{c \vdash_{\mathcal{C}} \exists Y(d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d'((c \vdash_{\mathcal{C}} \exists Y(d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash d')}{\left(X; c; \forall \boldsymbol{x}(e \xrightarrow{l} a), \Gamma\right) \rightarrow_{\mathrm{LCC}^\ell} (X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \Gamma)}$$

$$\textsc{Persistent Ask (with labeling)}\quad \frac{c \vdash_{\mathcal{C}} \exists Y(d \otimes e[\boldsymbol{t}/\boldsymbol{x}]) \qquad Y \cap \mathrm{fv}(X, c, \Gamma) = \varnothing \qquad \forall d'((c \vdash_{\mathcal{C}} \exists Y(d' \otimes e[\boldsymbol{t}/\boldsymbol{x}])) \wedge (d' \vdash_{\mathcal{C}} d) \Rightarrow d \dashv\vdash d')}{\left(X; c; \forall \boldsymbol{x}(e \overset{l}{\Rightarrow} a), \Gamma\right) \rightarrow_{\mathrm{LCC}^\ell} \left(X \cup Y; d; a[\boldsymbol{t}/\boldsymbol{x}], \forall \boldsymbol{x}(e \overset{l}{\Rightarrow} a), \Gamma\right)}$$

Agents A are translated to a particular family of labeled agents denoted A^{ℓ_0} with the labeling transformation, which ensures the following conditions: each label carries a distinct symbol taken from a set $\mathcal{P}$ of fresh predicate symbols and the arguments enumerate exactly the free variables of the ask. Such a labeling is simple to obtain as soon as $\mathcal{P}$ is large enough to label each ask of a.

Example 5. The dining philosophers (example 2) can be labeled as follows:

$$\begin{aligned}
&\forall N(\,\mathrm{diner}(N) \overset{p_1}{\Longrightarrow}\\
&\quad \exists K(\forall I(\,\mathrm{recphilo}(K, I) \overset{p_2(K,N)}{\Longrightarrow}\\
&\qquad \mathrm{fork}(K, I) \parallel\\
&\qquad \exists J.(\,J \textbf{ is } (I+1) \bmod N \parallel\\
&\qquad\quad (\,\mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J) \overset{p_3(I,J,K)}{\Longrightarrow}\\
&\qquad\qquad \mathrm{eat}(K, I) \parallel (\mathrm{eat}(K, I) \xrightarrow{p_4(I,J,K)} \mathrm{fork}(K, I) \otimes \mathrm{fork}(K, J)) \parallel\\
&\qquad\quad (I < N - 1 \xrightarrow{p_5(I,J,K,N)} \mathrm{recphilo}(K, J)) \parallel\\
&\qquad \mathrm{recphilo(K, 0)})))))
\end{aligned}$$

4.2 The ask-lifting transformation

The ask-lifting transformation is defined with two helper functions. $\langle a\rangle^{\mathcal{C}}$ transforms the agent a to constraints where asks become linear tokens. $\langle a\rangle^{\forall}$ puts in parallel every ask occurring in a and the representing token is added to the guard. A persistent ask restores the token, a transient ask consumes it.

The function $\langle\cdot\rangle^{\mathcal{C}} : A^{\ell} \to \mathcal{C}$ is defined inductively as follows:

$$\left\langle \forall \boldsymbol{x}(c \xrightarrow{f(\boldsymbol{t})} a)\right\rangle^{\mathcal{C}} = f(\boldsymbol{t}) \qquad \left\langle \forall \boldsymbol{x}(c \xRightarrow{f(\boldsymbol{t})} a)\right\rangle^{\mathcal{C}} = f(\boldsymbol{t}) \qquad \left\langle \exists x.a\right\rangle^{\mathcal{C}} = \exists x.\left\langle a\right\rangle^{\mathcal{C}}$$

$$\left\langle a \parallel b\right\rangle^{\mathcal{C}} = \left\langle a\right\rangle^{\mathcal{C}} \otimes \left\langle b\right\rangle^{\mathcal{C}} \qquad\qquad \left\langle c\right\rangle^{\mathcal{C}} = c$$

The function $\langle\cdot\rangle^{\forall} : A^{\ell_0} \to A^{\forall}$ is defined inductively as follows:

$$\left\langle \forall \boldsymbol{x}(c \xrightarrow{f(\boldsymbol{v})} a)\right\rangle^{\forall} = \forall \boldsymbol{v}\boldsymbol{x}(f(\boldsymbol{v}) \otimes c \xRightarrow{f(\boldsymbol{v})} \left\langle a\right\rangle^{\mathcal{C}}) \parallel \left\langle a\right\rangle^{\forall}$$

$$\left\langle \forall \boldsymbol{x}(c \xRightarrow{f(\boldsymbol{v})} a)\right\rangle^{\forall} = \forall \boldsymbol{v}\boldsymbol{x}(f(\boldsymbol{v}) \otimes c \xRightarrow{f(\boldsymbol{v})} f(\boldsymbol{v}) \otimes \left\langle a\right\rangle^{\mathcal{C}}) \parallel \left\langle a\right\rangle^{\forall}$$

$$\left\langle \exists x.a\right\rangle^{\forall} = \left\langle a\right\rangle^{\forall} \qquad \left\langle a \parallel b\right\rangle^{\forall} = \left\langle a\right\rangle^{\forall} \parallel \left\langle b\right\rangle^{\forall} \qquad \left\langle c\right\rangle^{\forall} = 1$$

The function $\langle\cdot\rangle^{\forall}$ is well-defined: every ask satisfies the side-condition for $A^{\forall}$.

Definition 14 (Ask-lifting). *The* agent ask-lifting function $[\![\cdot]\!]^{\uparrow} : A \to A^{\uparrow}$ *transforms the agent a to the agent $[\![a]\!]^{\uparrow} = \langle a^{\ell}\rangle^{\forall} \parallel \langle a^{\ell}\rangle^{\mathcal{C}}$ where a is translated to a^{ℓ} by the labeling defined in 4.1 with symbol predicates from a subset $\mathcal{P}$ of $\mathcal{P}_c$ whose predicates do not appear in a. $[\![\cdot]\!]^{\uparrow}$ is well-defined as soon as the set $\mathcal{P}$ is large enough to label agent a.*

Main Result 3 (Bisimilarity) *Let a be a labeled* LCC *agent. Let $\sim \subseteq \mathcal{K} \times \mathcal{K}$ be the relation such that $\kappa \sim \kappa'$ if and only if $\kappa \equiv_{\mathrm{L}} (X; c; \Gamma)$ is $\Rightarrow$-accessible from a and $\kappa' \equiv_{\mathrm{L}} (X; c \otimes \langle\Gamma\rangle^{\mathcal{C}}; \langle a\rangle^{\forall})$. Then, $\sim$ is a bisimilarity.*

Corollary 3 (Semantics preservation). *For every* LCC *agent a:*

- *for all $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; \Gamma)$ from a, there is a $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c'; \langle a\rangle^{\forall})$ from $[\![a]\!]^{\uparrow}$ such that $\exists X(c \otimes \langle\Gamma\rangle^{\mathcal{C}}) \dashv\vdash_{\mathcal{C}} \exists X'(c')$;*
- *for all $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c'; \langle a\rangle^{\forall})$ from $[\![a]\!]^{\uparrow}$, there exists a $\Rightarrow_{\mathrm{L}}$-accessible configuration $(X; c; \Gamma)$ from a and $\exists X(c \otimes \langle\Gamma\rangle^{\mathcal{C}}) \dashv\vdash_{\mathcal{C}} \exists X'(c')$.*

Example 6. The labeled diner (example 5) can be lifted as follows:

$$\begin{array}{ll}
\forall N(& p_1 \otimes \mathrm{diner}(N) \Rightarrow p_1 \otimes p_2(K,N) \otimes \mathrm{recphilo}(K,0)) \parallel \\
\forall IKN(& p_2(K,N) \otimes \mathrm{recphilo}(K,I) \Rightarrow \\
 & p_2(K,N) \otimes \mathrm{fork}(K,I) \otimes \\
 & \exists J(J \textbf{ is } (I+1) \bmod N \otimes p_3(I,J,K) \otimes p_5(I,J,K,N))) \parallel \\
\forall IJK(& p_3(I,J,K) \otimes \mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J) \Rightarrow \\
 & p_3(I,J,K) \otimes \mathrm{eat}(K,I) \otimes p_4(I,J,K)) \parallel \\
\forall IJK(& p_4(I,J,K) \otimes \mathrm{eat}(K,I) \Rightarrow \mathrm{fork}(K,I) \otimes \mathrm{fork}(K,J)) \parallel \\
\forall IJKN(& p_5(I,J,K,N) \otimes I < N-1 \Rightarrow \mathrm{recphilo(K,J)}) \parallel p_1
\end{array}$$

4.3 Encoding the λ-calculus in CHR

The following transformation from pure λ-terms to LCC is proved correct [13]. Every function, *aka* λ-value, is represented by a variable K. The constraint $\mathrm{apply}(K, X, V)$ represents that V should code the result of the application of the function (coded by) K to the λ-term (coded by) X. Therefore, the transformation of a λ-abstraction $\lambda x.e$ coded by K should be a persistent ask which transforms, for all X and V, the constraint $\mathrm{apply}(K, X, V)$ to the equality constraint between V and the evaluation of $e[t/x]$, where t is the λ-term coded by X. The equality constraint is put at the level of λ-variables. The constraint $\mathrm{value}(K)$ indicates that the λ-term K has been reduced to a value so as to encode the particular call-by-value strategy [26].

Definition 15 (Call-by-value λ-calculus in LCC [13]). *For every λ-term e, $[\![e]\!]$ is a function from variables to* LCC *agents. $[\![e]\!]$ is described inductively on the structure of e:*

- $[\![X]\!](K) = \langle X = K \otimes \mathrm{value}(K) \rangle$
- $[\![\lambda X.e]\!](K) = \forall XV(\mathrm{apply}(K, X, V) \otimes \mathrm{value}(X) \Rightarrow [\![e]\!](V) \parallel \mathrm{value}(X))$
- $[\![f\ e]\!](K) = \exists XY(\mathrm{apply}(X, Y, K) \parallel [\![f]\!](X) \parallel [\![e]\!](Y))$

Each ask introduced by this transformation corresponds to a λ-abstraction and this property is preserved by ask-lifting. Therefore, the CSR program obtained by translation has one rule for each λ-abstraction.

We explicit below the direct transformation from λ-terms to CSR. We suppose that the labeling has been prepared directly in λ-terms: λ-abstractions are of the form $\lambda_i X.e$ where i is a unique index.

Definition 16 (Call-by-value λ-calculus in CHR). *For every λ-term e, $[e]$ is a function from variables to pairs* CHR *programs and queries, each component being denoted $[e]^p$ and $[e]^g$. $[e]$ is described inductively on the structure of e as follows.*

- $[X](K) \quad = (\ \varnothing \quad ;(X = K, value(K)))$
- $[\lambda X_i.e](K) = (\ [e]^p(V) \cup \{$
  $\langle p_i(K, \boldsymbol{v}), \mathrm{value}(X), \mathrm{apply}(K, X, V) \Leftrightarrow p_i(K, \boldsymbol{v}), \mathrm{value}(X), [e]^g(V). \rangle$
  $\}$ $\quad ; p_i(K, \boldsymbol{v}))$
  where $\boldsymbol{v} = \mathrm{fv}(\lambda X.e)$ *and X and V fresh variables*
- $[f\ e](K) \quad = (\ [f]^p(X) \cup [f]^p(Y)\ ;([f]^g(X), [e]^g(Y), \mathrm{apply}(X, Y, K)))$
  where X and Y fresh variables

$p_i(\boldsymbol{v})$ CHR *constraints are supposed to be fresh. Then, the* CSR *program associated to e is $P[e] = [e]^p(R) \cup \{\langle start(R, \boldsymbol{v}) \Leftrightarrow [e]^g(R). \rangle\}$ and the query is $q[e] = \mathrm{start}(R, \boldsymbol{v})$ with $\boldsymbol{v} = \mathrm{fv}(e)$.*

It is immediate that the program and the goal produced by the transformation above correspond syntactically to the composition of the three transformations: λ-terms to LCC (definition 15) to flat-LCC (definition 14) to CSR (definition 12). Therefore, the transformation preserves the semantics as composition of semantics preserving transformations.

In the case of a CHR encoding, the rule associated to each λ-abstraction can be denoted as a simpagation: $\langle p_i(K, \boldsymbol{v}), \mathrm{value}(X) \backslash \mathrm{apply}(K, X, V) \Leftrightarrow [e]^g(V). \rangle$.

Example 7. The λ-term $(\lambda_1 X.\lambda_2 Y.X)\ A\ B$ is transformed to the rules:

```
start(R,A,B) ⇔
      p1(F1), apply(F1,A0,F2), apply(F2,B0,R),
      A=A0, value(A0), B=B0, value(B0).
p1(F1), value(X) \ apply(F1,X,F2) ⇔ p2(F2,X).
p2(F2,X), value(Y) \ apply(F2,Y,R) ⇔ X = R, value(R).
```

and the following goal, where the variable R codes the result:

```
| ?- start(R, A, B).
p1(_) value(_X) value(_) value(_X) p2(_, _X)
R = A
```

5 Conclusion

The translations presented in this paper generalize previous links between CHR and linear logic. As the work for modules in LCC suggest[20], variables and CHR constraints are expressive enough to embed a form of closures, and thus leads to a simple encoding for the λ-calculus.

Whereas the state during a CHR derivation is entirely determined by the contents of constraint stores, an LCC configuration contains suspended agents as well. The *ask-lifting* transformation reveals that suspensions can be reified to linear tokens, which in turns become CHR constraints: transient asks are consumed whereas persistent asks are propagated.

Behaviors of programs or agents obtained by translation are precisely related to their antecedents by (strong) bisimulation. To our knowledge, only weak bisimulation results [27] were formulated in the literature for CHR before. To achieve strong bisimulation in our case, we have managed to circumscribe collaterally the non-determinism in the naive operational semantics of CHR and in the operational semantics of LCC.

Future work

Suggested transformations are straightforward enough to be implemented. However, the moot point is to understand the relevance of CHR refined semantics for the translated LCC agents: the question of control in LCC is still open.

Interpreting operational semantics (indifferently CHR or LCC) as a proof search method in linear logic reveals a parallel between the elimination of solving non-determinism and focalization theory [28] which remains to explore.

Transition systems considered here are non-labeled: this was sufficient for semantics preservation and there are good intuitions about the pair of involved firing rules at each step. Formalizing these intuitions by labeling with rule names seems feasible but with low interest. However, labels usually serve to follow messages that an agent either sends or receives. A challenge would be to label

⇒-transitions by constraints whereas each single transition consumes some while adding others.

The closure encoding may suggest a new programming style, complementary to the imperative RAM-based style recently described [29]. Optimization of the CHR constraints which reify closures could be explored.

Acknowledgments. I would like to thank François Fages, Rémy Haemmerlé, Julien Martin and Sylvain Soliman for all the useful discussions and comments since the very beginning of this work.

References

1. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
2. Coquery, E., Fages, F.: From typing constraints to typed constraint systems in CHR. In: Proceedings of Third workshop on Rule-based Constraint Reasoning and Programming, associated to CP'01. (2001)
3. Sulzmann, M., Wazny, J., Stuckey, P.J.: A framework for extended algebraic data types. In: In Proc. of FLOPS'06, volume 3945 of LNCS, Springer-Verlag (2006) 47–64
4. Otzbeyer, H., , Pretschner, E.: Testing concurrent reactive systems with constraint logic programming. In: In Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming. (2000)
5. Abdennadher, S., Marte, M.: University course timetabling using constraint handling rules. Journal of Applied Artificial Intelligence **14** (2000)
6. Saraswat, V.A.: Concurrent constraint programming. ACM Doctoral Dissertation Awards. MIT Press (1993)
7. Saraswat, V.: A brief introduction to linear concurrent constraint programming. Xerox PARC (1993)
8. Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: operational and phase semantics. Information and Computation **165** (2001) 14–41
9. Girard, J.Y.: Linear logic. Theoretical Computer Science **50(1)** (1987)
10. Betz, H., Frühwirth, T.W.: A linear-logic semantics for constraint handling rules. In: Proceeding of CP 2005, 11th. (2005) 137–151
11. Giusto, C., Gabbrielli, M., Meo, M.C.: Expressiveness of multiple heads in CHR. In: SOFSEM '09: Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, Berlin, Heidelberg, Springer-Verlag (2009) 205–216
12. Haemmerlé, R., Betz, H.: Verification of constraint handling rules using linear logic phase semantics. In: Proceeding of CHR 2008, the fifth Constraint Handling Rules Workshop. Report Series 08-10, RICS-Linz (2008)
13. Haemmerlé, R.: Fermetures et Modules dans les Langages Concurrents avec Contraintes fondés sur la Logique Linéaire. PhD thesis, Univ. Paris 7. Soutenance le 17 janvier 2008 (2007)
14. Johnsson, T.: Lambda lifting: transforming programs to recursive equations. In: Proc. of a conference on Functional programming languages and computer architecture, New York, NY, USA, Springer-Verlag New York, Inc. (1985) 190–203

15. Anders Schack-Nielsen, C.S.: Invited talk: The CHR-Celf connection. In Frühwirth, T., Schrijvers, T., eds.: Proceedings of the fifth Constraint Handling Rules Workshop CHR'08. (2008)
16. Schack-Nielsen, A., Schürmann, C.: Celf — a logical framework for deductive and concurrent systems (system description). In: IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning, Berlin, Heidelberg, Springer-Verlag (2008) 320–326
17. Best, E., de Boer, F.S., Palamidessi, C.: Concurrent constraint programming with information removal. In: Proceedings of Coordination. Lecture Notes in Computer Science, Springer-Verlag (1997)
18. Dijkstra, E.: Hierarchical ordering of sequential processes. Acta Informatica **1** (1971) 115–138
19. Abdennadher, S., Frühwirth, T.W., Meuss, H.: Confluence and semantics of constraint simplification rules. Constraints **4** (1999) 133–165
20. Haemmerlé, R., Fages, F., Soliman, S.: Closures and modules within linear logic concurrent constraint programming. In Arvind, V., Prasad, S., eds.: Proceedings of FSTTCS 2007, IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science. Volume 4855 of Lecture Notes in Computer Science., Springer-Verlag (2007) 544–556
21. Jaffar, J., .Maher, M., .Marriott, K., Stuckey, P.: The semantics of constraint logic programs. Journal of Logic Programming **37** (1998) 1–46
22. Warren, D.S.: The Andorra principle. In: Presented at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden. (1988)
23. Sangiorgi, D.: On the bisimulation proof method. Mathematical Structures in Computer Science **8** (1998) 447–479
24. Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for constraint handling rules. In: PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2005) 218–229
25. Duck, G.J., Stuckey, P.J., Banda, M.G.D.L., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: In 20th International Conference on Logic Programming (ICLP'04, Springer (2004) 90–104
26. Plotkin, G.: Call-by-name, call-by-value and the lambda calculus. Theoretical Computer Science **1** (1975) 125–159
27. Koninck, L.D.: Logical algorithms meets CHR: A meta-complexity theorem for Constraint Handling Rules with rule priorities. Theory and Practice of Logic Programming (to appear) (2009)
28. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2** (1992)
29. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. In: Proceedings of the second Constraint Handling Rules Workshop, at ICLP'05. (2005) 3–17

Equivalence of CHR States Revisited

Frank Raiser, Hariolf Betz, and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. While it is generally agreed-upon that certain classes of CHR states should be considered equivalent, no standard definition of equivalence has ever been established. Furthermore, the compliance of equivalence with rule application is generally assumed, but has never been proven. We systematically develop an axiomatic notion of state equivalence based on rule applicability and the declarative semantics. We supply the missing proof for its compliance with rule application and provide a proof technique to determine equivalence of given states. The compliance property leads to a simplified formulation of the operational semantics. Furthermore, it justifies a novel view based on equivalence classes of states which provides a powerful proof technique.

1 Introduction

While equivalence of states is apparently an elementary concept in Constraint Handling Rules (CHR), the community has never agreed on a standard definition of that concept up to now. A plethora of definitions of state equivalence has been introduced in various areas of application. For example, the operational equivalence algorithm compares two resulting states of different programs for equivalence [1]. Equivalence is the basis for invariants such as in [2]. Several definitions [3–6] have been introduced in the context of confluence considerations. Finally, from an operational point of view, it is clear that the normalization function of the operational semantics implicitly assumes a notion of state equivalence.

As the various authors had different intentions, the resulting definitions of state equivalence vary considerably. There is a general agreement that from an operational point of view any notion of state equivalence should be compliant with rule applications, i.e. for equivalent states the same rules are applicable and lead to equivalent results. However, this property has never been proven for any of the previously proposed definitions. Another general agreement is that from a declarative point of view the logical reading of equivalent states should also be equivalent.

Our aim is therefore to develop a definition of state equivalence that satisfies both the operational and the declarative view. Instead of defining a notion of state equivalence for a fixed problem setting, we intend a notion for which these generally agreed-upon properties hold. By construction, our definition of state equivalence then is compliant with rule application and the logical reading of

states. Thus, it becomes a generic proof technique that can be applied to specific problems with the additional knowledge that the above-mentioned properties are satisfied.

In this paper, we make the following contributions:

- We justify a set of desirable properties for a general notion of state equivalence and present them in the form of example cases in Sect. 2.2.
- We give a concise overview of the existing definitions of state equivalence in Sect. 2.3 and compare their behavior with respect to our example cases in Sect. 2.4. We show that none of the existing definitions satisfies all of the example cases.
- We introduce an axiomatic definition of state equivalence in Sect. 3.1 along with several useful properties following from that definition.
- We present a necessary, sufficient, and decidable criterion for determining equivalence of states in Sect. 3.2.
- In Sect. 3.3 we show that our definition of state equivalence complies with all of the example cases defined in Sect. 2.2.
- In Sect. 4.1, we show that our notion of equivalence leads to a clearer definition of the operational semantics. We prove its equivalence to the traditional definition and – for the first time in the literature – we show that state equivalence is indeed compliant with rule application.
- In Sect. 4.3, we present a view of the CHR transition system that is based on equivalence classes of states rather than individual states.

Finally, we present our conclusions and establish possible further research paths in Sect. 5.

2 Existing Equivalence Definitions

Constraint Handling Rules (CHR) [7–9] is a concurrent committed-choice rule-based programming language, originally developed as a portable language extension for the implementation of user-defined constraint solvers. In the main part of our paper, specific knowledge of CHR is not required. For a discussion of the operational semantics of CHR, refer to Sect. 4.

In this section, we evaluate existing definitions of state equivalence and postulate desirable properties of an equivalence relation over CHR states. To this end, we present several prototypical example cases of equivalent and non-equivalent CHR states in Sect. 2.2. In Sect. 2.3 we concisely introduce the different notions of state equivalence that have been proposed so far, before we investigate how these notions apply to our example states in Sect. 2.4. We first define the syntax of CHR states and rules. The corresponding transition system is given in Sect. 4 where we apply our results from Sect. 3.

2.1 Preliminaries

In CHR, we distinguish two disjoint sets of constraints which we call *built-in constraints* and *CHR constraints*.

Definition 1 (CHR State). *A* CHR state σ *is a tuple* $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$. *The* goal $\mathbb{G}$ *is a multiset of CHR constraints. The* built-in constraint store $\mathbb{B}$ *is a conjunction of built-in constraints.* $\mathbb{V}$ *is a set of* global variables.

We use $\sigma, \sigma_0, \sigma_1, \ldots$ *to denote states and* Σ *to denote the set of all states.*

We found more elaborate definitions than Def. 1 unsuitable for our demand on the logical reading of equivalent states, because they contain information that is not reflected in the logical reading of the states, such as the propagation history [10]. Depending on the domain of application, our result can be extended to these definitions. Our notion of state clearly separates the three components that each have to be treated differently by state equivalence.

For any CHR state we distinguish three sets of variables according to the following definition.

Definition 2 (Variable Types). *For the variables occurring in a state* $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ *we distinguish three different types:*

1. *a variable* $v \in \mathbb{V}$ *is called a* global *variable*
2. *a variable* $v \notin \mathbb{V}$ *is called a* local *variable*
3. *a variable* $v \notin (\mathbb{V} \cup \mathbb{G})$ *is called a* strictly local *variable*

The following definition introduces the logical reading of CHR states.

Definition 3 (Logical Reading of CHR States).

Let σ *be a CHR state of the form* $\langle \{g_1, g_2, \ldots g_n\}, \mathbb{B}, \mathbb{V} \rangle$. *Then the logical reading of* σ *is:*

$$\bar{\exists}_{\mathbb{V}} \left(g_1 \wedge g_2 \wedge \ldots \wedge g_n \wedge \mathbb{B} \right)$$

where $\bar{\exists}_{\mathbb{V}}$ *is existential quantification of all free variables except those in* $\mathbb{V}$.

A CHR program defines a set of rules, as given in Def. 4, by which the constraints in $\mathbb{G}$ are to be rewritten to a final solved form. The store $\mathbb{B}$ contains built-in constraints that have been posted to an underlying solver. These are assumed to be solved implicitly by the host language $\mathcal{H}$ according to a complete and decidable constraint theory $\mathcal{CT}$. The set $\mathbb{V}$ usually contains the variables that occur in the initial state of a computation and can be thought of as communication channels with the outside world.

Definition 4 (CHR rule).

A CHR rule *is of the following form*

$$H_1 \backslash H_2 \Leftrightarrow G | B_c \uplus B_b$$

where the head H_1, H_2 *consists of two multisets of CHR constraints, the* guard G *is a conjunction of built-in constraints, and the body consists of a multiset* B_c *of CHR constraints and a conjunction* B_b *of built-in constraints.*

2.2 Examples of CHR States

Let us now consider the following examples of equivalent and non-equivalent states to highlight the differences between existing definitions of state equivalence:

$$\langle c(X), \top, \emptyset\rangle \equiv \langle c(Y), \top, \emptyset\rangle \quad (1)$$
$$\langle c(X), X = 0, \{X\}\rangle \equiv \langle c(0), X = 0, \{X\}\rangle \quad (2)$$
$$\langle \top, X \geq 0 \wedge X \leq 0 \wedge Y = 0, \{X\}\rangle \equiv \langle \top, X = 0, \{X\}\rangle \quad (3)$$
$$\langle c(0), \top, \{X\}\rangle \equiv \langle c(0), \top, \emptyset\rangle \quad (4)$$
$$\langle c(X), \top, \{X\}\rangle \not\equiv \langle c(Y), \top, \{Y\}\rangle \quad (5)$$

The equivalences (1)-(3) are motivated by the fact that the same rules are applicable to these states with the same results.

As the states in equivalence (4) have the same logical reading $c(0)$ according to Def. 3 we require them to be equivalent. Note that unused global variables can practically occur, for example when applying rule $c(X) \Leftrightarrow c(0)$ to the state $\langle c(X), \top, \{X\}\rangle$. Concerning non-equivalence (5), note that X, Y are free variables and therefore the logical readings $c(X), c(Y)$ are not equivalent.

2.3 Existing Definitions

Over the last decade, the CHR community proposed various definitions for state equivalence. The following list identifies six distinct categories of equivalence definitions in the literature:

I The definitions based on variable renaming [1, 4, 11, 12] are often as simple as stating that two states are equivalent (or variants) if they can be obtained by variable renaming only. These definitions arose from the notion of variance on terms.

II In [13] a definition is given that is based on renaming of local variables as well as logical equivalence of built-in stores.

III In [5] a similar definition is given for arbitrary binary relations rather than for CHR states only.

IV [14] gives another definition based on the so-called refined operational semantics [15] of CHR.

V [6] – a follow-up to [14] – extends the definition with the usage of a unifier instead of variable renaming.

VI In [16, 17] a *normalization function* is defined. While we emphasize that this definition was not targeted towards determining state equivalence, we include it in this work due to its clear structure that is similar to our proposed definition. When we talk about equivalence with respect to normalization we implicitly assume that two states are equivalent iff their normalizations are syntactically equivalent.

2.4 Comparison of Existing Equivalence Definitions

We have applied each of the existing definitions to each of the example cases. The results are presented in Table 1. Each entry shows whether the two corresponding example states are considered equivalent or not, according to the category used in that row. The last row presents the results that we deem desirable. As we can see, none of the previously published definitions of state equivalence respects all of the example cases.

	(1)	(2)	(3)	(4)	(5)
Def. I	$\equiv$	$\not\equiv$	$\not\equiv$	$\not\equiv$	$\equiv$
Def. II	$\equiv$	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$
Def. III	$\equiv$	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$
Def. IV	$\equiv$	$\not\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Def. V	$\equiv$	$\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Def. VI	$\not\equiv$	$\equiv$	$\equiv$	$\not\equiv$	$\not\equiv$
Desired	$\equiv$	$\equiv$	$\equiv$	$\equiv$	$\not\equiv$

Table 1. Comparison of different state equivalence definitions

3 An Axiomatic Definition of Equivalence

In this section, we introduce an axiomatic definition of equivalence that satisfies all the desirable properties we identified in the previous section. We present our definition in Sect. 3.1 along with several properties. In Sect. 3.2 we give a necessary, sufficient, and decidable criterion to prove equivalence and non-equivalence between CHR states. In Sect. 3.3, we prove compliance with the example cases from Sect. 2.2.

3.1 Definition of State Equivalence

Our notion of state equivalence is given in the following definition.

Definition 5 (State Equivalence).
Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions:

1. (Equality as Substitution)
$$\langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x/t]\,, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$$
2. (Transformation of the Constraint Store) *If $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:*
$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}, \mathbb{B}', \mathbb{V} \rangle$$

3. (Omission of Non-Occurring Global Variables) *If X is a variable that does not occur in $\mathbb{G}$ or $\mathbb{B}$ then:*

$$\langle \mathbb{G}, \mathbb{B}, \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$$

4. (Equivalence of Failed States)

$$\langle \mathbb{G}, \bot, \mathbb{V} \rangle \equiv \langle \mathbb{G}', \bot, \mathbb{V} \rangle$$

The axioms are chosen such that we can guarantee compliance with the operational semantics (cf. Sect. 4.1) as well as the logical readings of CHR states.

Firstly, names of local variables in CHR states are chosen non-deterministically upon execution. Hence, considering these names invariant with respect to state equivalence suggests itself. In combination, axiom 1 and axiom 2 guarantee this desired property (cf. Lemma 1:1).

Axiom 1 and axiom 2 are furthermore invariant with respect to rule applicability and comply with logical equivalence of the logical readings. The same holds for axiom 4: On the logical level, inconsistent logical readings are of course logically equivalent. It is necessary to guarantee compliance with the operational semantics as we justify in Sect. 4.2.

Axiom 3 suggests itself with regard to logical readings, since adding or removing global constraints results in syntactically identical and therefore indistinguishable logical readings. Operationally, unused global variables have no effect, so it stands to reason to consider them redundant.

Lemma 1 states several properties that follow from Def. 5.

Lemma 1 (Properties of State Equivalence). *The equivalence relation over CHR states given in Def. 5 has the following properties:*

1. (Renaming of Local Variables) *Let x, y be variables such that $x, y \notin \mathbb{V}$ and y does not occur in $\mathbb{G}$ or $\mathbb{B}$:*

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x/y]\,, \mathbb{B}\,[x/y]\,, \mathbb{V} \rangle$$

2. (Partial Substitution) *Let $\mathbb{G}\,[x \wr t]$ be a multiset where* some *occurrences of x are substituted with t:*

$$\langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x \wr t]\,, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$$

3. (Logical Equivalence) *If*

$$\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle \equiv \langle \mathbb{G}', \mathbb{B}', \mathbb{V}' \rangle$$

then $\mathcal{CT} \models \exists \bar{y}.\mathbb{G} \wedge \mathbb{B} \leftrightarrow \exists \bar{y}'.\mathbb{G}' \wedge \mathbb{B}'$, where $\bar{y}, \bar{y}'$ are the local variables of $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \langle \mathbb{G}', \mathbb{B}', \mathbb{V}' \rangle$, respectively.

Proof.

Property 1: *By transformation of the constraint store, we have that* $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ *is equivalent to* $\langle \mathbb{G}, x \doteq y \wedge \mathbb{B}, \mathbb{V} \rangle$*. We apply equality as substitution and get* $\langle \mathbb{G}[x/y], x \doteq y \wedge \mathbb{B}, \mathbb{V} \rangle$ *which by transformation is equivalent to* $\langle \mathbb{G}[x/y], \mathbb{B}[x/y], \mathbb{V} \rangle$*.*

Property 2: *By substitution, we have that both* $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ *and* $\langle \mathbb{G}[x \wr t], x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$ *are equivalent to* $\langle \mathbb{G}[x/t], x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$*. The equivalence relation is implicitly symmetric and transitive.*

Property 3: *All conditions given in Def. 5 correspond to valid logical equivalences:*

Definition 5:1 *preserves logical equivalence since*

$$\mathbb{G} \wedge x \doteq t \leftrightarrow \mathbb{G}[x/t] \wedge x \doteq t$$

Definition 5:2: *As* $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ *and the variables in* $\bar{s}, \bar{s}'$ *do not occur in* $\mathbb{G}, \mathbb{G}'$*, we have*

$$\mathcal{CT} \models \exists \bar{y}.\mathbb{B} \wedge \mathbb{G} \leftrightarrow \exists \bar{y}'.\mathbb{B}' \wedge \mathbb{G}$$

Definition 5:3: *For a variable* X *that does not occur in* $\mathbb{B}$ *or* $\mathbb{G}$ *we obviously have*

$$\mathcal{CT} \models \exists X.\exists \bar{y}.\mathbb{B} \wedge \mathbb{G} \leftrightarrow \exists \bar{y}.\mathbb{B} \wedge \mathbb{G}$$

Definition 5:4 *preserves logical equivalence due to the* ex falso quodlibet *property.*

As logical equivalence is reflexive, transitive, and symmetric, Prop. 3 holds. □

3.2 A Sufficient and Decidable Criterion for State Equivalence

Logical equivalence between $\exists \bar{y}.\mathbb{G} \wedge \mathbb{B}$ and $\exists \bar{y}'.\mathbb{G}' \wedge \mathbb{B}'$ is a necessary but not a sufficient condition for state equivalence between $\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle$ and $\langle \mathbb{G}', \mathbb{B}', \mathbb{V}' \rangle$ (cf. Lemma 1:3). This is due to the fact that unlike logical equivalence, state equivalence preserves the multiplicities of logically equivalent user-defined constraints. A similar condition which is also sufficient can be formulated in *linear logic* [18].

Theorem 1 gives a necessary and sufficient criterion for deciding state equivalence. Note that due to its preconditions it technically decides a smaller relation than $\equiv$, because it only applies to the case that local variables are renamed apart and the set of global variables is unchanged.

However, this restriction is not problematic for deciding equivalence in general. By Lemma 1 we are free to rename local variables apart and by Def. 5:3 we can adjust the sets of global variables to match. Therefore, Thm. 1 gives us a necessary and sufficient criterion for equivalence of arbitrary states: first we transform the states into equivalent states that satisfy the preconditions, then we apply the theorem. The transformation is straightforward and equivalence-preserving, hence, the result we get from the theorem applies to the original states by transitivity of $\equiv$. Finally, decidability of our criterion is a direct consequence of decidability of $\mathcal{CT}$.

Theorem 1 (Criterion for $\equiv$). *Let $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle$ be CHR states with local variables $\bar{y}, \bar{y}'$ that have been renamed apart.*

$$\sigma \equiv \sigma' \text{ iff } \mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

Proof. Let $\mathcal{C}$ be a binary predicate on CHR states such that $\mathcal{C}(\langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle)$ holds iff

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

$\Rightarrow$:

We show that each of the three implicit conditions – reflexivity, symmetry and transitivity – as well as all the four explicit conditions of Def. 5 are sound w.r.t. criterion $\mathcal{C}$.

Reflexivity: *Reflexivity is given as the following judgment is obviously true:*

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B})) \wedge \forall(\mathbb{B} \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}))$$

Symmetry: *Symmetry of $\mathcal{C}$ is obvious.*

Transitivity: *Assume three states $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle, \sigma'' = \langle \mathbb{G}'', \mathbb{B}'', \mathbb{V} \rangle$ with distinct local variables $\bar{y}, \bar{y}', \bar{y}''$ such that $\mathcal{C}(\sigma, \sigma')$ and $\mathcal{C}(\sigma', \sigma'')$. By definition, we have:*

$$\begin{array}{ll} \mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) & (i) \\ \mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B})) & (ii) \\ \mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists \bar{y}''.((\mathbb{G}' = \mathbb{G}'') \wedge \mathbb{B}'')) & (iii) \\ \mathcal{CT} \models \forall(\mathbb{B}'' \rightarrow \exists \bar{y}'.((\mathbb{G}' = \mathbb{G}'') \wedge \mathbb{B}')) & (iv) \end{array}$$

From (i) and (iii) follows:

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}''.((\mathbb{G} = \mathbb{G}'') \wedge \mathbb{B}''))$$

From (ii) and (iv) follows:

$$\mathcal{CT} \models \forall(\mathbb{B}'' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}'') \wedge \mathbb{B}))$$

Consequently, $\mathcal{C}(\sigma, \sigma'')$.

Equality as Substitution: *Assume two states $\sigma = \langle \mathbb{G}, x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}[x/t], x \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$ with local variables $\bar{y}, \bar{y}'$. As $\mathcal{CT} \models \forall(x \doteq t \rightarrow (\mathbb{G} = \mathbb{G}[x/t]))$, we have $\mathcal{C}(\sigma, \sigma')$.*

Transformation of the Constraint Store: *Assume two states $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}, \mathbb{B}', \mathbb{V} \rangle$ with local variables $\bar{y}, \bar{y}'$ and strictly local variables $\bar{s}, \bar{s}'$ such that $CT \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$. This implies the following judgment:*

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}) \wedge \mathbb{B}))$$

Hence, $\mathcal{C}(\sigma, \sigma')$.

Omission of Non-Occurring Global Variables: *Does not apply since σ, σ' share the set $\mathbb{V}$ of global variables.*

Equivalence of Failed States: *For any two failed states, we have states of the form $\langle \mathbb{G}, \bot, \mathbb{V} \rangle, \langle \mathbb{G}', \bot, \mathbb{V} \rangle$. The following judgment proves $\mathcal{C}(\langle \mathbb{G}, \bot, \mathbb{V} \rangle, \langle \mathbb{G}', \bot, \mathbb{V} \rangle)$:*

$$\mathcal{CT} \models \forall(\bot \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \bot)) \wedge \forall(\bot \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \bot))$$

$\Leftarrow$*:*

We consider two CHR states $\sigma = \langle \mathbb{G}, \mathbb{B}, \mathbb{V} \rangle, \sigma' = \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle$ with local variables $\bar{y}$ and $\bar{y}'$. We assume that

$$\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')) \wedge \forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$$

If there does not exist a pairwise matching $\mathbb{G} = \mathbb{G}'$, we have $\mathbb{B} = \mathbb{B}' = \bot$, which proves that $\sigma \equiv \sigma'$ by Def. 5:4. In the following, we assume that a pairwise matching $\mathbb{G} = \mathbb{G}'$ does exist.

It follows from $\forall(\mathbb{B} \rightarrow \exists \bar{y}'.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}')$ by Def. 5:2 that:

$$\sigma \equiv \langle \mathbb{G}, \mathbb{G} = \mathbb{G}' \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$$

By Def. 5:1 we have:

$$\sigma \equiv \langle \mathbb{G}', \mathbb{G} = \mathbb{G}' \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$$

From $\forall(\mathbb{B}' \rightarrow \exists \bar{y}.((\mathbb{G} = \mathbb{G}') \wedge \mathbb{B}))$ we get by Def. 5:2 that:

$$\sigma \equiv \langle \mathbb{G}', \mathbb{B}', \mathbb{V} \rangle = \sigma'$$

□

3.3 Example Cases Revisited

We claimed earlier that our definition of state equivalence replicates the desired equivalences and non-equivalences given in the examples in Sect. 2.2. Now we revisit these examples and prove their compliance with our definition.

Example (1)**:** $\langle c(X), \top, \emptyset \rangle \equiv \langle c(Y), \top, \emptyset \rangle$

Proof. Renaming of local variables (cf. Lemma 1:1) □

Example (2)**:** $\langle c(X), X = 0, \{X\} \rangle \equiv \langle c(0), X = 0, \{X\} \rangle$

Proof. Follows directly from Def. 5:1. □

Example (3)**:** $\langle \top, X \geq 0 \wedge X \leq 0 \wedge Y = 0, \{X\} \rangle \equiv \langle \top, X = 0, \{X\} \rangle$

Proof. Follows from Def. 5:2, as:

$$\mathcal{CT} \models \exists Y.(X \geq 0 \wedge X \leq 0 \wedge Y = 0) \leftrightarrow (X = 0)$$

□

Example (4)**:** $\langle c(0), \top, \{X\} \rangle \equiv \langle c(0), \top, \emptyset \rangle$

Proof. Follows directly from Def. 5:3. □

Example (5)**:** $\langle c(X), \top, \{X\} \rangle \not\equiv \langle c(Y), \top, \{Y\} \rangle$

Proof. Follows from Thm. 1, as $\mathcal{CT} \not\models \forall(\top \rightarrow c(X) = c(Y))$. □

4 Impact on the Operational Semantics

In this section, we discuss the impact of our definition of state equivalence on the operational semantics of CHR. Section 4.1 applies our notion of state equivalence to the traditional operational semantics. The resulting formulation is clearer and more lucid than the traditional one. More importantly, it enables us to prove that state equivalence is indeed compliant with rule applications. This important property – while generally assumed – has never been proven before. This in turn gives rise to a definition of the operational semantics based directly on equivalence classes of states which we present in Sect. 4.3.

4.1 A Simplified Formulation of the Operational Semantics

In this section, we present a formulation of the operational semantics based on state equivalence. Our definition is not only based on the traditional definition, but is also provably equivalent. Consider the following definition for the traditional operational semantics, adjusted from [9]:

Definition 6 (Traditional Operational Semantics). *For a CHR program $\mathcal{P}$, the state transition system $(\Sigma, \mapsto)$ is defined as follows, where $(r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b)$ is a copy of a rule in $\mathcal{P}$ containing only fresh variables.*

$$\frac{\begin{array}{c}(r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b) \text{ with fresh variables } \bar{y} \\ \mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists \bar{y}.(H_1 = H_1' \wedge H_2 = H_2' \wedge G))\end{array}}{\langle H_1' \uplus H_2' \uplus \mathbb{G}, \mathbb{B}, \mathbb{V}\rangle \mapsto^r \langle H_1' \uplus B_c \uplus \mathbb{G}, H_1 = H_1' \wedge H_2 = H_2' \wedge G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle}$$

Integrating the notion of state equivalence permits removing the matching that has traditionally been hidden in the complex formula $H_1 = H_1' \wedge H_2 = H_2'$. Furthermore, imposing a guard condition on $\mathcal{CT}$ becomes dispensable, leading to the following simplified operational semantics:

Definition 7 (Operational Semantics). *For a CHR program $\mathcal{P}$ we define the state transition system $(\Sigma, \rightarrowtail)$ as follows, where $(r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b)$ is a copy of a rule in $\mathcal{P}$ containing only fresh variables.*

$$\frac{(r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b)}{\langle H_1 \uplus H_2 \uplus \mathbb{G}, G \wedge \mathbb{B}, \mathbb{V}\rangle \rightarrowtail^r \langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle}$$

$$\frac{\sigma' \equiv \sigma \quad \sigma \rightarrowtail^r \tau \quad \tau \equiv \tau'}{\sigma' \rightarrowtail^r \tau'}$$

If the rule r is clear from the context or any rule is sufficient we simply write $\sigma \rightarrowtail \tau$. As usual $\rightarrowtail^*$ is the reflexive-transitive closure of $\rightarrowtail$.

Theorem 2 proves the equivalence of both definitions. An important aspect of this equivalence, and therefore the second rule of Def. 7, is that state equivalence is compliant with rule applications. It seems that the CHR community intuitively agrees this property holds for state equivalence. It is noteworthy that, to the best of our knowledge, the following is effectively the first published proof for this important property.

Theorem 2 (Equivalence of the Definitions). *For a CHR state σ we have*

1. *If $\sigma \rightarrowtail^r \tau$ then there exists a state $\tau' \equiv \tau$ with $\sigma \mapsto^r \tau'$*
2. *If $\sigma \mapsto^r \tau'$ then there exists a state $\tau \equiv \tau'$ with $\sigma \rightarrowtail^r \tau$*

Proof.

1: *Let $r \ @ \ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b$, $\sigma = \langle H_1 \uplus H_2 \uplus \mathbb{G}, G \wedge \mathbb{B}, \mathbb{V}\rangle \rightarrowtail^r \tau = \langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle$, and $\sigma' \equiv \sigma$. Let $\bar{y}, \bar{y}'$ be the local variables of σ, σ' respectively.*
Since r uses only fresh variables, we can assume w.l.o.g. that the local variables of σ are renamed apart from the local variables of σ' and σ' is of the form $\langle H'_1 \uplus H'_2 \uplus \mathbb{G}', \mathbb{B}', \mathbb{V}\rangle$.
As $\sigma \equiv \sigma'$ we get by Thm. 1 that

$$\mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists\bar{y}.((H'_1 \uplus H'_2 \uplus \mathbb{G}' = H_1 \uplus H_2 \uplus \mathbb{G}) \wedge G \wedge \mathbb{B}))$$

and consequently:

$$\mathcal{CT} \models \forall(\mathbb{B}' \rightarrow \exists\bar{y}.(H'_1 = H_1 \wedge H'_2 = H_2 \wedge G))$$

Therefore, we can apply the traditional operational semantics and get:

$$\sigma' \mapsto^r \langle H'_1 \uplus B_c \uplus \mathbb{G}', H_1 = H'_1 \wedge H_2 = H'_2 \wedge G \wedge B_b \wedge \mathbb{B}', \mathbb{V}\rangle = \tau'$$

By the above and Def. 5:2 we get:

$$\tau' \equiv \langle H'_1 \uplus B_c \uplus \mathbb{G}', H_1 = H'_1 \wedge H_2 = H'_2 \wedge \mathbb{G} = \mathbb{G}' \wedge G \wedge B_b \wedge \mathbb{B}' \wedge \mathbb{B}, \mathbb{V}\rangle$$

By Def. 5:1 and Def. 5:2 we then get:

$$\tau' \equiv \langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle = \tau$$

2: *Let $\sigma = \langle H'_1 \uplus H'_2 \uplus \mathbb{G}, \mathbb{B}, \mathbb{V}\rangle \mapsto^r \tau' = \langle H'_1 \uplus B_c \uplus \mathbb{G}, H_1 = H'_1 \wedge H_2 = H'_2 \wedge G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle$.*
As $\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \exists\bar{y}.(H_1 = H'_1 \wedge H_2 = H'_2 \wedge G))$ we apply Def. 5:2 to σ:

$$\sigma \equiv \langle H'_1 \uplus H'_2 \uplus \mathbb{G}, H_1 = H'_1 \wedge H_2 = H'_2 \wedge G \wedge \mathbb{B}, \mathbb{V}\rangle$$

Using Def. 5:1 we get by substitution:

$$\sigma \equiv \langle H_1 \uplus H_2 \uplus \mathbb{G}, H_1 = H'_1 \wedge H_2 = H'_2 \wedge G \wedge \mathbb{B}, \mathbb{V}\rangle$$

We can apply rule r according to Def. 7 now resulting in

$$\sigma \rightarrowtail^r \langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge (H_1 = H'_1 \wedge H_2 = H'_2 \wedge \mathbb{B}), \mathbb{V}\rangle = \tau$$

By Def. 5:1 we get $\tau \equiv \tau'$. □

4.2 Termination on Failure

Both formulations of the operational semantics given above allow arbitrary rule applications on failed states. In the case of the traditional operational semantics, the applicability condition is of the form $\mathcal{CT} \models \forall(\mathbb{B} \rightarrow \ldots)$. This condition is trivially satisfied for any inconsistent built-in store due to the principle *ex contradictio quodlibet* (ECQ). In our formulation of the operational semantics arbitrary rule applications are due to the explicit equivalence of failed states established in Def. 5:4. For example, under a program $P = \{a \Leftrightarrow b\}$ both $\langle c, \bot, \emptyset\rangle \mapsto^* \langle b, \bot, \emptyset\rangle$ and $\langle c, \bot, \emptyset\rangle \rightarrowtail^* \langle b, \bot, \emptyset\rangle$ are correct.

For theoretical considerations such as correctness analyses, this property is intentional: Firstly, it corresponds to the ECQ property which holds in most logical formalisms. Secondly, allowing derivations from failed states preserves monotonicity with respect to strengthening the constraint store such that e.g. $\langle \mathbb{G}, \mathbb{B}, \mathbb{V}\rangle \mapsto^* \langle \mathbb{G}', \mathbb{B}', \mathbb{V}'\rangle$ implies $\langle \mathbb{G}, \mathbb{B} \wedge \bar{\mathbb{B}}, \mathbb{V}\rangle \mapsto^* \langle \mathbb{G}', \mathbb{B}' \wedge \bar{\mathbb{B}}, \mathbb{V}'\rangle$, regardless of whether $\mathbb{B} \wedge \bar{\mathbb{B}}$ is consistent.

In a practical implementation, however, this property would lead to trivial non-termination of any failed computation. Existing implementations consequently do not allow computation from failed states. Formally, this property can be captured by introducing $\mathcal{CT} \models \exists\mathbb{B}$ as an applicability condition. Alternatively, we can solve this issue by explicitly disallowing atomic derivation steps between equivalent states. Unlike the first solution, such a formulation also avoids trivial non-termination originating from rules of the form $H \Leftrightarrow G \mid H$.

4.3 Founding the Operational Semantics on Equivalence Classes

Having finally shown the compliance of state equivalence with rule application, we revisit the operational semantics once more: We know that a rule application is possible for all states equivalent to a state containing the head and guard of that rule and that we are free to choose any element of the set of equivalent result states. Therefore, the actual syntactical representation of a state is of no importance, leading to a different view on the transition system: instead of considering all syntactical representations as different states, we propose the following reformulation of the operational semantics based on equivalence classes over states.

Definition 8. *For a CHR program $\mathcal{P}$ we define the state transition system $(\Sigma/\equiv\ ,\ \rightarrowtail)$ as follows. The application of a rule r is based on a copy of it that contains only fresh variables.*

$$\frac{r\ @\ H_1 \setminus H_2 \Leftrightarrow G \mid B_c \uplus B_b}{[\langle H_1 \uplus H_2 \uplus \mathbb{G}, G \wedge \mathbb{B}, \mathbb{V}\rangle] \rightarrowtail^r [\langle H_1 \uplus B_c \uplus \mathbb{G}, G \wedge B_b \wedge \mathbb{B}, \mathbb{V}\rangle]}$$

This concise definition of the operational semantics of CHR requires only one rule and combines all of our results on state equivalence and its behavior with regard to rule applications. Furthermore, it leads to simplifications in other research areas where state equivalence is relevant. For example, confluence analysis

in CHR is made complicated by different syntactical representations. Figure 1 depicts that term rewriting systems (TRS) require all computations to reach the exact same final term. However, the corresponding CHR results demand equivalence of final states. Under the operational semantics as given in Def. 8, the original *diamond property* of confluence depicted in Fig. 1 found in the TRS literature applies directly to CHR.

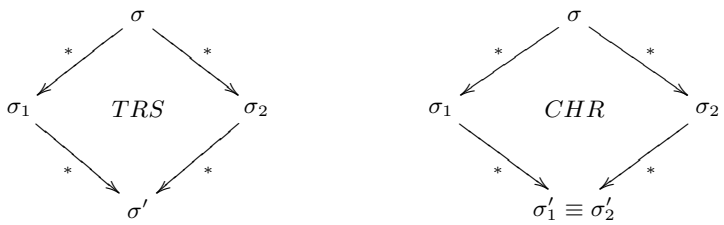

Fig. 1. Diamond property of confluence in TRS and CHR

5 Conclusion

As the main result, we have proven the compliance of state equivalence with rule application. To the best of our knowledge, this is the first published proof of this important property for any notion of equivalence.

This property has significant impact on the formulation of the operational semantics of CHR: It allows for a considerably more compact and lucid definition of the operational semantics than the ones known in the literature. Furthermore, it justifies an operational semantics of CHR based on equivalence classes of states rather than individual states.

We have presented the first axiomatic definition of state equivalence in the literature. It is more intuitive than existing definitions and provides elegant proof techniques such as applied in our proof of Lemma 1. However, it is not always trivial to prove state equivalence – and more so: non-equivalence – of arbitrary CHR states using the axiomatic definition. Therefore, we have presented a necessary, sufficient, and decidable criterion for both equivalence and non-equivalence of states. Example case (5) in Sect. 3.3 shows an application of this criterion.

We have evaluated the previously published definitions of state equivalence for a set of example cases and shown that none of them satisfies all of the examples. Contrarily, our definition of state equivalence satisfies all of the example cases.

We expect that the notion of state extensions [6] leads to a further enhanced formulation of the operational semantics. This concept would allow to disregard parts of the state that are unaffected by rule application. This would lead to an even more compact definition of the state transition system.

As we have shown, the operational semantics based on equivalence classes allows a simplified formulation of confluence results for CHR. We furthermore plan to reinvestigate results from observable confluence, operational equivalence, and their combination under this context. Combined with the expressive possibilities of state extensions this should lead to more concise formulations and proofs of all those results.

References

1. Abdennadher, S., Frühwirth, T.: Operational equivalence of CHR programs and constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming, CP 1999. Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57
2. Raiser, F., Frühwirth, T.: Strong joinability analysis for graph transformation systems in CHR. In: 5th International Workshop on Computing with Terms and Graphs, TERMGRAPH'09. (2009)
3. Abdennadher, S., Frühwirth, T.W., Meuss, H.: On confluence of constraint handling rules. In: Principles and Practice of Constraint Programming, 2nd International Conference. (1996) 1–15
4. Frühwirth, T., Pierro, A.D., Wiklicky, H.: Probabilistic constraint handling rules. In: Electronic Notes in Theoretical Computer Science. (2002) 1–16
5. Haemmerlé, R., Fages, F.: Abstract critical pairs and confluence of arbitrary binary relations. In: RTA '07: Proc. 18th Intl. Conf. Term Rewriting and Applications. Volume 4533 of Lecture Notes in Computer Science., Paris, France, Springer-Verlag (2007)
6. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for constraint handling rules. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. Volume 4670 of Lecture Notes in Computer Science., Porto, Portugal, Springer-Verlag (2007) 224–239
7. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
8. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (2003)
9. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009) draft.
10. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Submitted to *Journal of Theory and Practice of Logic Programming* (2008)
11. Meister, M.: Efficient Declarative Programming. PhD thesis, Ulm University, Ulm, Germany (2007)
12. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, University of Melbourne, Australia (2005)
13. Raiser, F., Tacchella, P.: On Confluence of Non-terminating CHR Programs. In Djelloul, K., Duck, G.J., Sulzmann, M., eds.: Constraint Handling Rules, 4th Workshop, CHR 2007, Porto, Portugal (2007) 63–76
14. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for constraint handling rules. In: Constraint Handling Rules, Third Workshop, CHR 2006. (2006)
15. Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: Logic Programming, 20th International Conference, ICLP 2004. (2004) 90–104

16. Abdennadher, S., Frühwirth, T., Meuss, H.: Confluence and semantics of constraint simplification rules. Constraints **4** (1999) 133–165
17. Abdennadher, S.: Rule-based Constraint Programming: Theory and Practice. Habilitationsschrift, Institute of Computer Science, LMU, Munich, Germany (2001)
18. Betz, H., Frühwirth, T.: A linear-logic semantics for Constraint Handling Rules. In: Principles and Practice of Constraint Programming, 11th International Conference, CP 2005. Volume 3709 of Lecture Notes in Computer Science., Sitges, Spain, Springer-Verlag (2005) 137–151

Operational Equivalence of Graph Transformation Systems

Frank Raiser and Thom Frühwirth

Faculty of Engineering and Computer Sciences, Ulm University, Germany
{Frank.Raiser|Thom.Fruehwirth}@uni-ulm.de

Abstract. Graph transformation systems (GTS) provide an important theory for numerous applications. With the growing number of GTS-based applications the comparison of operational equivalence of two GTS becomes an important area of research. This work introduces a notion of operational equivalence for graph transformation systems. The embedding of GTS in constraint handling rules (CHR) provides the basis for a decidable and sufficient criterion for operational equivalence of GTS. It is based on the operational equivalence test for CHR programs. A direct application of adapting this test to GTS allows automatic removal of redundant rules.

1 Introduction

Graph transformation systems (GTS) describe complex structures and systems in an expressive and versatile way. The principal idea of graph transformation systems is to apply graph production rules to a host graph. This involves finding a subgraph that matches the rule's graph and that is modified according to that rule.

As more applications based on graph transformations emerge it is becoming an important area of research to compare two GTS with each other. This comparison can be used to check whether two GTS solve the same problem, to verify that an optimized version of a GTS adheres to a non-optimized, but simpler, specification-GTS, to remove redundant rules, and more.

Different mechanisms for rewriting graphs have been developed [1] and in this work we make use of the so-called DPO approach [2]. It provides a sound category theoretical basis for modeling graph transformations.

Example 1. In Fig. 1 a graph transformation system consisting of two rules is shown. The graphical notation is explained in more detail later. The two rules replace edges of type a by edges of type b. An application of the operational equivalence, presented in Sect. 3.1, is the automatic removal of the second rule due to its redundancy.

In [3] an embedding of GTS in constraint handling rules (CHR) [4, 5] is given. In the context of CHR, operational equivalence is already an active area of research [6]. Therefore, we apply the results on operational equivalence in CHR

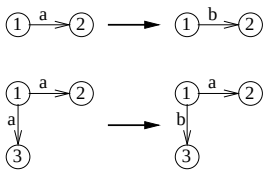

Fig. 1. Graph transformation system – the second rule is redundant

to embedded graph transformation systems. This gives us a first notion of operational equivalence for GTS and at the same time a decidable and sufficient criterion for it. In CHR research a successful application of the operational equivalence test is the removal of redundant rules [7, 8]. By building upon the GTS embedding and operational equivalence of CHR, we show that this application can be adapted to remove redundant GTS rules as well.

Note that there is an orthogonal notion for *behavioral equivalence* of graph transformation systems [9]. Behavioral equivalence investigates the behavior of models, i.e. host graphs, which are transformed into new models while preserving behavior as given by a semantics based on another graph transformation system. Similarly, in [10] bisimilarity for GTS is discussed. While bisimilarity originated from process calculi and is focused on the transitions made during computation of a result, our approach only compares the final computation results independently of how they are reached.

This paper is a preliminary work for establishing the notion of operational equivalence on graph transformation systems. We make the following contributions:

- We present definitions of joinability and operational equivalence in GTS transferred from CHR research.
- We show that given our existing embedding of GTS into CHR allows to reuse the operational equivalence test as a sufficient criterion for operational equivalence of two GTS.
- We apply the previous result to automatically remove redundant rules from a GTS.
- We observe the problem of a direct formulation of our criterion in the GTS context.

This work is divided into the following sections: Section 2 presents the necessary preliminaries for graph transformation systems, constraint handling rules, the embedding of GTS in CHR, and operational equivalence in CHR. We then introduce operational equivalence for GTS in Sect. 3, before concluding in Sect. 4.

2 Preliminaries

In this section we introduce the necessary preliminaries for this work. As our work is derived from results in different research areas, each of the following sections introduces a particular research topic. In Sect. 2.1 we introduce Graph Transformation Systems (GTS) and in Sect. 2.2 Constraint Handling Rules (CHR). Section 2.3 discusses the embedding of a GTS in CHR. Finally, Sect. 2.4 highlights existing achievements on operational equivalence for CHR programs.

2.1 Graph Transformation Systems

The following definitions for graphs and graph transformation systems have been adapted from [2].

Definition 1 (type graph, typed graph).
A graph $G = (V, E, \mathrm{src}, \mathrm{tgt})$ *consists of a finite set* V *of nodes, a finite set* E *of edges and two morphisms* $\mathrm{src}, \mathrm{tgt} : E \to V$ *specifying source and target of an edge, respectively. A* type graph TG *is a graph with unique labels for all nodes and edges.*

For multiple graphs we refer to the node set V *of a graph* G *as* V_G *and analogously for edge sets and the* $\mathrm{src}, \mathrm{tgt}$ *morphisms. We further define the degree of a node as* $\deg : V \to \mathbb{N}, v \mapsto \#\{e \in E \mid \mathrm{src}(e) = v\} + \#\{e \in E \mid \mathrm{tgt}(e) = v\}$. *When the context graph is clear the subscript is omitted.*

A typed graph G *is a tuple* $(V, E, \mathrm{src}, \mathrm{tgt}, \mathrm{type}, TG)$ *where* $(V, E, \mathrm{src}, \mathrm{tgt})$ *is a graph,* TG *a type graph, and* type *a morphism with* $\mathrm{type} = (\mathrm{type}_V, \mathrm{type}_E)$ *and* $\mathrm{type}_V : V \to V_{TG}, \mathrm{type}_E : E \to E_{TG}$. *The* type *morphism is a graph morphism, therefore, it has to satisfy the following condition:* $\forall e \in E : \mathrm{type}_V(\mathrm{src}(e)) = \mathrm{src}_{TG}(\mathrm{type}_E(e)) \wedge \mathrm{type}_V(\mathrm{tgt}(e)) = \mathrm{tgt}_{TG}(\mathrm{type}_E(e))$

For the purpose of simplicity, the above definition avoids an additional label morphism in favor of identifying variable names with labels. As there are often multiple graphs containing the same node due to inclusion morphisms we use $\deg_G(v)$ to specify the degree of a node v with respect to the graph G.

Example 2. Figure 2 shows an example for a type graph and a corresponding typed graph. The type graph at the bottom is used to define a node type and two edge types a and b. The typed graph at the top left assigns a unique node type (or edge type) to each node (or edge) via the type morphism represented by the dotted lines. The typed graph at the top right uses a shorter notation that implicitly defines the type morphism by specifying the corresponding labels.

Definition 2 (GTS, rule).
A Graph Transformation System *(GTS) is a tuple consisting of a type graph and a finite set of graph production rules. A* graph production rule *– also simply called* rule *if the context is clear – is a tuple* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *of graphs* $L, K,$ *and* R *with inclusion morphisms* $l : K \to L$ *and* $r : K \to R$.

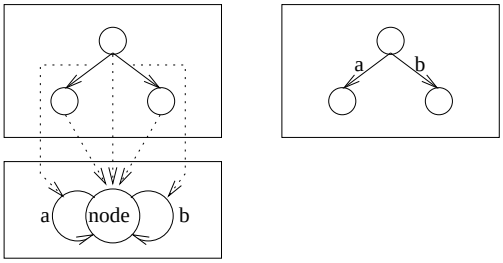

Fig. 2. Example of a type graph and typed graphs

We distinguish two kinds of typed graphs: *rule graphs* and *host graphs*. Rule graphs are the graphs L, K, R of a graph production rule p and host graphs are graphs to which the graph production rules can be applied. We, furthermore, make use of graph transformations based on the double-pushout approach (DPO) as defined in [2]. Most notably, we require a so-called match morphism $m : L \to G$ to apply a rule p to a typed host graph G. In this work we only consider injective match morphisms (see [3]). The transformation yielding the typed graph H is written as $G \overset{p,m}{\Longrightarrow} H$. H is given mathematically by constructing D as shown in Fig. 3, such that (1) and (2) are pushouts in the category of typed graphs [2]. Intuitively, the graph L on the left-hand side is matched as a subgraph of G and its occurrence in G is then replaced by the right-hand side graph R. The intermediate graph K is the context graph, which contains the nodes and edges in both, L and R, i.e. all nodes and edges matched to K remain unchanged during the transformation.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & {\scriptstyle k}\downarrow & (2) & \downarrow{\scriptstyle n} \\ G & \xleftarrow{f} & D & \xrightarrow{g} & H \end{array}$$

Fig. 3. Double-pushout approach

The application of a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ to G consists of transforming G into H by performing the construction of D and H such that (1) and (2) in Fig. 3 are pushouts. A more implementation-oriented interpretation of a rule application is that all nodes and edges in $m(L \setminus l(K))$ are removed from G to create $D = (G \setminus m(L)) \cup m(l(K))$ and then all nodes and edges in $n(R \setminus r(K))$ are added to create $H = D \cup n(R \setminus r(K))$.

Example 3. In Fig. 1 two graph production rules are shown in their shorthand notation. The numbers for the nodes imply the l and r morphisms and the graph K is implied as well and consists of the nodes with no edges for the first

rule and a single edge of type a for the second rule. The first rule when applied to a graph G replaces an edge of type a by an edge of type b, whereas the second rule makes a similar replacement within a larger context.

2.2 Constraint Handling Rules (CHR)

This section presents the syntax and operational semantics of constraint handling rules [5]. Intuitively, CHR is a rule-based multiset rewriting system. For this work we consider a subset of CHR by considering only one kind of rule and omitting guards. For the complete possibilities of CHR see [4, 5].

The constraints manipulated by CHR are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are handled by a constraint solver while user-defined constraints are defined by a CHR program. In this work the required built-in constraints are syntactic equalities and basic arithmetics.

Simplification rules are of the form

$$Rulename \ @ \ H_1, \ldots, H_i \Leftrightarrow B_1, \ldots, B_k$$

where *Rulename* is an optional unique identifier of a rule, the head $H = H_1, \ldots, H_i$ is a non-empty conjunction of user-defined constraints, and the body $B = B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints. Note that we make sloppy use of the terms conjunction, sequence, and multiset with respect to $H_1, \ldots, H_i$ and $B_1, \ldots, B_k$.

The operational semantics is based on an underlying *constraint theory* $\mathcal{CT}$ for the built-in constraints and a *state*, which is a tuple $\langle G, C, \mathcal{V} \rangle$ where G is a goal store, i.e. a multiset of user-defined constraints, C is a conjunction of built-in constraints, and $\mathcal{V}$ is the set of global variables. The variables in $\mathcal{V}$ are also called variables-of-interest and, intuitively, differ from other variables occurring in the state by being considered universally quantified [5]. When comparing different states we make use of an equivalence relation $\equiv$ on CHR states. This equivalence accounts for different syntactical representations, including renaming of local variables, equality substitutions, and logically equivalent built-in stores.

A simplification rule of the form $H \Leftrightarrow B$ is *applicable* to a state $\langle E \cup G, C, \mathcal{V} \rangle$ if $CT \models \forall(C \rightarrow \exists \overline{x}(H = E))$ where $\overline{x}$ are the variables in H and $=$ is syntactic equality. The above condition intuitively corresponds to finding a subset E of constraints in the state that, together with the built-in store C, match the head of a rule ($H = E$). We then define the following state transition for the application of the rule: $\langle E \cup G, C, \mathcal{V} \rangle \rightarrowtail \langle B_u \cup G, (H = E) \wedge C \wedge B_b, \mathcal{V} \rangle$ where $B = B_u \cup B_b$ with B_u being user-defined and B_b being built-in constraints. As usual, $\rightarrowtail^*$ denotes the reflexive-transitive closure of the $\rightarrowtail$ relation. When considering multiple programs $\rightarrowtail_{\mathcal{P}}$ denotes a transition based on a rule from program $\mathcal{P}$.

2.3 GTS in CHR

In this section we present how a graph transformation system can be embedded into CHR based on previous work in [3]. In order to embed a GTS in CHR, we

have to encode its graph production rules as CHR rules and provide a conjunction of goal constraints corresponding to the host graph. We then recapitulate the soundness and completeness results of our embedding.

For encoding a GTS in CHR we first determine the constraints needed for encoding the rules and host graph based on the type graph:

Definition 3 (Type Graph Encoding).
For a type graph $\mathcal{TG}$ we define the set $\mathcal{C}$ of required constraints to encode graphs typed over $\mathcal{TG}$ as the minimal set including $v/2 \in \mathcal{C}$ for $v \in V_{\mathcal{TG}}$ and $e/3 \in \mathcal{C}$ for $e \in E_{\mathcal{TG}}$.

Example 4 (cont). For our example of the GTS, every node in the typed graph has the same type and we have two edge types. Based on this we need the following constraints: $\mathrm{n}/2, \mathrm{a}/3, \mathrm{b}/3$

We assume all nodes and edges of the type graph TG to be uniquely labeled such that the introduced constraints have unique names as well. Note that this is no restriction on the typed graphs as there can be any number of nodes or edges of the same type. These constraints allow us to encode typed graphs. The following definition of `chr` distinguishes between $\texttt{chr}(\mathrm{host}, \cdot)$ and $\texttt{chr}(\mathrm{keep}, \cdot)$ which intuitively correspond to host and rule graphs.

Definition 4 (Typed Graph Encoding).
For a typed graph G based on a type graph TG the set of constraints encoding G is defined differently for host and rule graphs. We define the following mappings for the encoding for an infinite set of variables VARS:

- $[\mathrm{type}_G(x)]$ *denotes the corresponding constraint name for encoding a node or edge of the given type.*
- $\mathrm{var} : G \to VARS, x \mapsto X_x$ *such that X_x is a unique variable associated to x, i.e.* var *is injective for the set of all graph nodes and edges.*
- $\mathrm{dvar} : G \to VARS, x \mapsto X_x$ *such that X_x is a unique variable associated to x, i.e.* dvar *is injective for the set of all graph nodes and edges.*

Using these mappings we define the following encoding of graphs:

$$\texttt{chr}_G(\mathrm{host}, x) = \begin{cases} [\mathrm{type}_G(x)](\mathrm{var}(x), \deg_G(x)) & \text{if } x \in V_G \\ [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{var}(\mathrm{src}(x)), \mathrm{var}(\mathrm{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

$$\texttt{chr}_G(\mathrm{keep}, x) = \begin{cases} [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{dvar}(x)) & \text{if } x \in V_G \\ [\mathrm{type}_G(x)](\mathrm{var}(x), \mathrm{var}(\mathrm{src}(x)), \mathrm{var}(\mathrm{tgt}(x))) & \text{if } x \in E_G \end{cases}$$

We make use of the notations $\texttt{chr}(\mathrm{host}, G) = \{\texttt{chr}_G(\mathrm{host}, x) \mid x \in G\}$ and $\texttt{chr}(\mathrm{keep}, G) = \{\texttt{chr}_G(\mathrm{keep}, x) \mid x \in G\}$. Furthermore, we omit the index G if the context is clear. For a node v encoded with $\texttt{chr}(\mathrm{keep}, v)$ we call $\mathrm{dvar}(v)$ *the* degree variable*.*

Example 5 (cont). When using the left-hand side graph of the first rule as a host graph G it is encoded in $\texttt{chr}(\mathrm{host}, G)$ as follows:

$$\mathrm{n}(N_1, 1), \mathrm{n}(N_2, 1), \mathrm{a}(E_1, N_1, N_2)$$

The same graph G occurring as a rule graph is encoded in $\mathtt{chr}(\mathrm{keep}, G)$ as follows:

$$\mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2)$$

We can now encode a complete graph production rule based on these definitions:

Definition 5 (GTS Rule in CHR).

For a graph production rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ from a GTS we define $\rho(p) = (C_L, C_R)$ with

- $C_L = \{\mathtt{chr}(\mathrm{keep}, x) \mid x \in K\} \cup \{\mathtt{chr}(\mathrm{host}, x) \mid x \in L \setminus K\}$
- $C_R = \{\mathtt{chr}(\mathrm{host}, x) \mid x \in R \setminus K\} \cup \{\mathtt{chr}(\mathrm{keep}, e) \mid e \in E_K\}$
  $\cup \{\mathtt{chr}(\mathrm{keep}, v'), \mathrm{var}(v) = \mathrm{var}(v'), \mathrm{dvar}(v') = \mathrm{dvar}(v) - \deg_L(v) + \deg_R(v) \mid v \in V_K\}$

The rule p is then encoded in CHR using $\rho(p) = (C_L, C_R)$ and in abuse of notation we use $\rho(p)$ for the CHR rule p @ $C_L \Leftrightarrow C_R$ as well as for the tuple (C_L, C_R).

Example 6 (cont.). As an example, consider the first rule from our example GTS, which replaces the a-edge by a b-edge. Its encoding as a CHR simplification rule is given below:

$$\begin{aligned} \mathrm{r1} \ @ \ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2) \\ & \Leftrightarrow \\ & \mathrm{n}(N_1', D_1'), N_1' = N_1, D_1' = D_1 - 1 + 1, \\ & \mathrm{n}(N_2', D_2'), N_2' = N_2, D_2' = D_2 - 1 + 1, \\ & \mathrm{b}(E_2, N_1, N_2) \end{aligned}$$

The above rule is created strictly according to Def. 5, but contains numerous superfluous constructs, like $D_1' = D_1 - 1 + 1$. Eliminating redundant expressions leads to the following simpler, yet equivalent, rule:

$$\begin{aligned} \mathrm{r1} \ @ \ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{a}(E_1, N_1, N_2) \\ & \Leftrightarrow \\ & \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{b}(E_2, N_1, N_2) \end{aligned}$$

For the soundness and completeness results of this embedding we have to make sure, that the CHR programs only work on valid encodings of graphs. While in CHR any combination of node and edge constraints can be considered as input, not all of those make sense in terms of a GTS. To avoid such problems – like inconsistent degree encodings or dangling edges – we make use of the following graph invariant. This invariant holds for all states that are the valid encoding of a corresponding graph.

Definition 6 (Invariant, Graph Invariant).
$\mathcal{I}(S)$ is a property such that for all S_0 and S_1, we have that if $S_0 \rightarrow S_1$ (or $S_0 \equiv S_1$) and $\mathcal{I}(S_0)$ holds then $\mathcal{I}(S_1)$ holds.
The graph invariant *$\mathcal{G}(\sigma)$ with $\sigma = \langle E, C, \mathcal{V}\rangle$ holds if there exist a graph G and a conjunction of equality constraints C', such that*

$$\langle E, C \wedge C', \emptyset\rangle \equiv \langle \mathtt{chr}(\mathrm{host}, G), \top, \emptyset\rangle.$$

For a state σ where $\mathcal{G}(\sigma)$ holds with a graph G we say σ is a $\mathcal{G}$-state based on G.

Another characterization of σ being a $\mathcal{G}$-state based on G is the expression $\sigma \equiv \langle \mathtt{chr}(\mathrm{keep}, G), C, \mathcal{V}\rangle$.

Using the notion of a $\mathcal{G}$-state based on G we can further identify the set of strong nodes. Those nodes are special to the CHR encoding of a GTS as the operational semantics of CHR ensures they cannot be deleted by rule applications. In [3] it is shown that they are the key to the strong joinability analysis of critical pairs, but we also need them here for the completeness result.

Definition 7 (Strong Nodes and Derivations).
For a CHR state $S = \langle \mathtt{chr}(\mathrm{keep}, G), C, \mathcal{V}\rangle$ which is a $\mathcal{G}$-state based on G we define the set of strong nodes *as: $\mathcal{S}(S) = \{v \in V_G \mid \mathrm{dvar}(v) = \deg_G(v) \notin C\}$*
A GTS derivation $G \overset{p,m}{\Longrightarrow} G'$ using $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is strong with respect to $S \subset V_G$ *if $\forall s \in S : s \in m(K) \vee s \notin m(L)$.*

Finally, we present the soundness and completeness results from [3]. The soundness result states that CHR computations correspond to strong derivations:

Theorem 1 (Soundness).
Let $\rho(p) = (C_L, C_R)$ be a rule applicable to $\sigma = \langle \mathtt{chr}(\mathrm{keep}, G), C, \mathcal{V}\rangle$ where $\mathcal{G}(\sigma)$ holds, such that $\sigma \rightarrowtail \sigma'$.
Then $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is applicable to G such that $G \overset{p,m}{\Longrightarrow} G'$ is strong w.r.t. $\mathcal{S}(\sigma)$. Furthermore, $\sigma' \equiv \langle \mathtt{chr}(\mathrm{keep}, G'), C', \mathcal{V}\rangle$ and $\mathcal{G}(\sigma')$ holds.

As mentioned previously the set of strong nodes cannot be deleted by CHR rule applications. Therefore, our completeness result is restricted such that only strong GTS derivations are possible in CHR. Note that this restriction becomes redundant if $\mathcal{S}(\sigma) = \emptyset$, which is especially true, when σ is a $\mathtt{chr}(\mathrm{host}, G)$-based encoding of a graph G.

Theorem 2 (Completeness).
Let $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, $G \overset{p,m}{\Longrightarrow} G'$, and let $\sigma = \langle \mathtt{chr}(\mathrm{keep}, G), C, \mathcal{V}\rangle$ be a $\mathcal{G}$-state based on G.
If $\forall x \in L \setminus K : m(x) \notin \mathcal{S}(\sigma)$, then $\rho(p) = (C_L, C_R)$ is applicable to σ. Furthermore, for $\sigma \rightarrowtail \sigma'$ it follows that $\sigma' \equiv \langle \mathtt{chr}(\mathrm{keep}, G'), C', \mathcal{V}\rangle$ and $\mathcal{G}(\sigma')$ holds.

2.4 Operational Equivalence in CHR

CHR is well-known for its decidable, sufficient, and necessary criterion for operational equivalence of terminating and confluent CHR programs [5, 6].

The understanding of operational equivalence within the CHR community intuitively means that the two programs should be able to compute equivalent outputs given the same input. Applied to a single state this behavior is called $\mathcal{P}_1, \mathcal{P}_2$-joinability:

Definition 8 ($\mathcal{P}_1, \mathcal{P}_2$-Joinable, Operational Equivalence).
Let $\mathcal{P}_1, \mathcal{P}_2$ be CHR programs. A state σ is $\mathcal{P}_1, \mathcal{P}_2$-joinable*, iff there are computations $\sigma \rightarrowtail^*_{\mathcal{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathcal{P}_2} \sigma_2$ with $\sigma_1 \equiv \sigma_2$ where all σ_i are final states with respect to $\mathcal{P}_i$.*
$\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent *iff all states σ are $\mathcal{P}_1, \mathcal{P}_2$-joinable.*

A decision algorithm for operational equivalence of CHR programs is presented in [6]. It is based on the analysis of critical states:

Definition 9 (Critical States).
Let $\mathcal{P}_1, \mathcal{P}_2$ be CHR programs. The set of critical states of $\mathcal{P}_1$ and $\mathcal{P}_2$ *is defined as* $\{\langle H, \top, \mathrm{vars}(H)\rangle \mid (H \Leftrightarrow B) \in \mathcal{P}_1 \cup \mathcal{P}_2\}$

Theorem 3 (Operational Equivalence via Critical States).
Let $\mathcal{P}_1, \mathcal{P}_2$ be terminating and confluent CHR programs. $\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent iff for all critical states σ of $\mathcal{P}_1$ and $\mathcal{P}_2$ it holds that σ is $\mathcal{P}_1, \mathcal{P}_2$-joinable.

3 Operational Equivalence of Graph Transformation Systems

In this section we introduce our notion of operational equivalence for GTS. Based on the embedding of GTS in CHR from [3] we use the existing decision algorithm from CHR for operational equivalence of two graph transformation systems.

As a basis we define the property of $\mathcal{S}_1, \mathcal{S}_2$-joinability for two graph transformation systems $\mathcal{S}_1, \mathcal{S}_2$. It is motivated by $\mathcal{P}_1, \mathcal{P}_2$-joinability in the context of CHR as given in Def. 8.

Definition 10 ($\mathcal{S}_1, \mathcal{S}_2$-joinability).
Let $\mathcal{S}_1, \mathcal{S}_2$ be two graph transformation systems. A typed graph G is $\mathcal{S}_1, \mathcal{S}_2$-joinable *iff there are derivations $G \Rightarrow^*_{\mathcal{S}_1} G_1$ and $G \Rightarrow^*_{\mathcal{S}_2} G_2$ with $G_1 \simeq G_2$ being final w.r.t. $\mathcal{S}_1$ and $\mathcal{S}_2$. Here $\simeq$ denotes traditional graph isomorphism and a graph G is considered* final *w.r.t. $\mathcal{S}$ if there is no transition $G \Rightarrow_{\mathcal{S}} H$ for any graph H.*

Building on $\mathcal{S}_1, \mathcal{S}_2$-joinability we can now define operational equivalence for graph transformation systems with the same intuitive understanding that two equivalent GTS should be able to produce the same result graphs up to isomorphism given an input graph:

Definition 11 (GTS Operational Equivalence).
Let $\mathcal{S}_1 = (\mathcal{P}_1, \mathcal{TG})$ and $\mathcal{S}_2 = (\mathcal{P}_2, \mathcal{TG})$ be two graph transformation systems. $\mathcal{S}_1, \mathcal{S}_2$ are operationally equivalent *iff for all graphs G typed over $\mathcal{TG}$ it holds that G is $\mathcal{S}_1, \mathcal{S}_2$-joinable.*

Similar to operational equivalence in CHR where it is futile to directly compare programs that use different constraints, Def. 11 requires $\mathcal{S}_1$ and $\mathcal{S}_2$ to be based on the same type graph $\mathcal{TG}$. With the previous results from [3] we can directly use CHR's operational equivalence as a sufficient criterion for deciding operational equivalence of two GTS:

Theorem 4 (CHR Operational Equivalence Implies GTS Operational Equivalence).
Let $\mathcal{S}_1, \mathcal{S}_2$ be graph transformation systems and $\mathcal{P}_1, \mathcal{P}_2$ their corresponding CHR programs. $\mathcal{S}_1, \mathcal{S}_2$ are operationally equivalent if $\mathcal{P}_1, \mathcal{P}_2$ are operationally equivalent.

Proof. Let G be a graph typed over $\mathcal{TG}$. Then the state $\sigma = \langle \mathtt{chr}(\mathrm{host}, G), \top, \emptyset \rangle$ is $\mathcal{P}_1, \mathcal{P}_2$-joinable by Def. 8. Therefore, there exist the final states $\sigma_1 \equiv \sigma_2$ with $\sigma \rightarrowtail^*_{\mathcal{P}_1} \sigma_1$ and $\sigma \rightarrowtail^*_{\mathcal{P}_2} \sigma_2$. By Thm. 1 we know that there exist corresponding derivations $G \Rightarrow^*_{\mathcal{S}_1} G_1$ and $G \Rightarrow^*_{\mathcal{S}_2} G_2$ such that σ_1 is a $\mathcal{G}$-state based on G_1 and σ_2 is a $\mathcal{G}$-state based on G_2. Due to Thm. 2 G_1 and G_2 are final states w.r.t. $\mathcal{S}_1$ and $\mathcal{S}_2$, and finally, the isomorphism between G_1 and G_2 is implied by $\sigma_1 \equiv \sigma_2$. Therefore, G is $\mathcal{S}_1, \mathcal{S}_2$-joinable. □

The reverse direction of Thm. 4 cannot be proved in a similarly simple way. This is due to the problem, that $\mathcal{P}_1, \mathcal{P}_2$-joinability is required for all states, including states that have no corresponding graph. By restricting observation to valid states we plan to derive a stronger characterization of operational equivalence of GTS (see Sect. 4).

3.1 Redundant Rule Removal

The redundant rule removal algorithm is an application of operational equivalence presented in [11]. It can be applied to graph transformation systems embedded in CHR. It requires the CHR program to be terminating and confluent with respect to states corresponding to graphs. This implies that the GTS is required to be confluent and terminating as well [3]. Any redundant rule of such a program then corresponds to a redundant rule for derivations of the GTS.

The basic idea of the algorithm is to try to remove a rule from the program and compare this modified program with the original program. If both are operationally equivalent the rule is redundant and can be removed. Note that depending on the order in which rules are tried different results are possible and this algorithm, hence does not guarantee to yield an operationally equivalent program with the minimal number of rules possible. Nevertheless, it proved to be an important algorithm especially in the context of automatically generated programs [7, 8].

Example 7. Consider again the GTS given in Fig. 1 and its embedding in CHR:

$$\begin{array}{ll} \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), & \Leftrightarrow \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \\ \mathrm{a}(E, N_1, N_2) & \quad \mathrm{b}(E', N_1, N_2) \end{array}$$

$$\begin{array}{ll} \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{n}(N_3, D_3), & \Leftrightarrow \mathrm{n}(N_1, D_1), \mathrm{n}(N_2, D_2), \mathrm{n}(N_3, D_3), \\ \mathrm{a}(E_1, N_1, N_2), \mathrm{a}(E_2, N_1, N_3) & \quad \mathrm{b}(E'_1, N_1, N_2), \mathrm{a}(E_2, N_1, N_3) \end{array}$$

The algorithm tries to remove the second rule from the program and then compare the two programs according to the operational equivalence test. The only relevant case is considering the head of the removed rule as input to both programs. The two computations may use different rules, but both programs compute a final state consisting of a graph with three nodes, an a-edge from the first to the third node, and an a-edge from the first to the second node.

The previous example shows that our approach uses strong derivations for testing operational equivalence. This is an important feature and as the following example demonstrates implementing our approach directly in GTS, that use non-strong derivations, is not possible.

Example 8. Consider the two graph transformation systems shown in Fig. 4. Using the graph on the left-hand sides of the two rules as input to both GTSs leads to isomorphic results as shown in Fig. 5 (1). However, assuming the slightly extended graph shown in Fig. 5 (2) instead gives two non-isomorphic result graphs. Therefore, only examining the critical states within the GTS using non-strong derivations is insufficient. Applying our approach to the GTS in Fig. 4, however, gives the intended result. We assume this difference is due to the isomorphism function and the track morphisms (see [12]) for the two GTS derivations being inconsistent (see also Sect. 4).

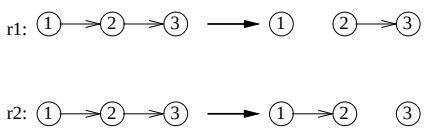

Fig. 4. Two operationally non-equivalent graph transformation systems

4 Conclusion

In this work we introduced operational equivalence of graph transformation systems. The proposed notion of operational equivalence is motivated by research in the context of CHR based on joinability of states. Interestingly, our previous work on embedding GTS in CHR [3] provides the means to directly derive

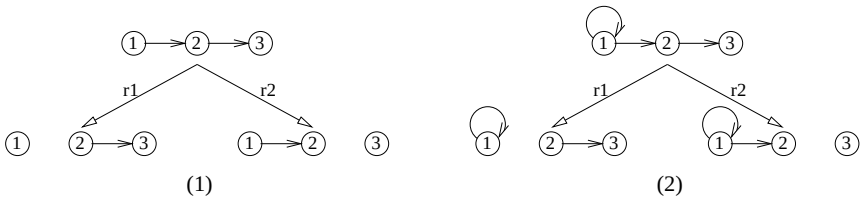

Fig. 5. Derivations for two initial graphs

a sufficient criterion for operational equivalence of two graph transformation systems.

As a direct application of this work we showed that our criterion is suitable for automatically removing redundant rules from a GTS. Finally, we have given an example demonstrating that a direct translation of our approach into the GTS context does not yield the expected outcome.

4.1 Future Work

As the final example showed for a GTS embedded in CHR nodes are implicitly tracked. We exploited this behavior earlier in [3] in order to decide strong joinability which in the GTS context is formulated via an explicit track morphism [12]. We hope to find a similar formulation of the operational equivalence test, in which an explicit check involving the track morphism allows us to reach a corresponding sufficient criterion within the GTS context.

The redundant rule used in the example in Fig. 1 in this work is *subsumed* by the other rule. This notion of subsumption for graph transformation systems is discussed by Kreowski and Valiente in [13] and an interesting future line of research is to compare our approach with that work. Because our approach relies on termination it cannot be as generic as their sufficient condition. However, to the best of our knowledge and as stated in [13] it remains open to find a verification procedure for that condition. Our approach might, hence, be able to verify redundancy for a subset of the rules that are redundant according to the criterion of Kreowski and Valiente.

Furthermore, in the context of CHR the notion of *operational c-equivalence* for a constraint c was introduced to extend the classes of programs that can be operationally equivalent. This notion is not suitable for the context of GTS as it corresponds to two graph transformation systems defined over two type graphs that share exactly one node. A more realistic case is the sharing of a subgraph of the type graphs which in turn gives rise to the idea of *operational C-equivalence* for a set of constraints C. Another reason to examine this also in the context of CHR is that a notion of operational C-equivalence is a generalization of both previous equivalence notions.

References

1. Blostein, D., Fahmy, H., Grbavec, A.: Practical use of graph rewriting. In: 5th Workshop on Graph Grammars and Their Application To Computer Science. Volume 1073 of Lecture Notes in Computer Science., Springer-Verlag (1995) 38–55
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006)
3. Raiser, F., Frühwirth, T.: Strong joinability analysis for graph transformation systems in CHR. In: 5th International Workshop on Computing with Terms and Graphs, TERMGRAPH'09. (2009)
4. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Accepted to *Journal of Theory and Practice of Logic Programming* (2009)
5. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009) to appear.
6. Abdennadher, S., Frühwirth, T.: Operational equivalence of CHR programs and constraints. In Jaffar, J., ed.: Principles and Practice of Constraint Programming, CP 1999. Volume 1713 of Lecture Notes in Computer Science., Springer-Verlag (1999) 43–57
7. Raiser, F.: Semi-automatic generation of CHR solvers for global constraints. In Stuckey, P.J., ed.: Principles and Practice of Constraint Programming, 14th International Conference, CP 2008. Volume 5202 of Lecture Notes in Computer Science., Sydney, Australia, Springer-Verlag (September 2008) 588–592
8. Abdennadher, S., Sobhi, I.: Generation of rule-based constraint solvers: Combined approach. In King, A., ed.: Logic-Based Program Synthesis and Transformation, 17th International Symposium, LOPSTR 2007, Kongens Lyngby, Denmark, August 23-24, 2007, Revised Selected Papers. Volume 4915 of Lecture Notes in Computer Science., Springer-Verlag (2007) 106–120
9. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: Proc. of ICGT '08 (International Conference on Graph Transformation). Volume 5214 of Lecture Notes in Computer Science., Springer-Verlag (2008) 242–256
10. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In Walukiewicz, I., ed.: Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004. Volume 2987 of Lecture Notes in Computer Science., Barcelona, Spain, Springer-Verlag (2004) 151–166
11. Abdennadher, S., Frühwirth, T.: Integration and optimization of rule-based constraint solvers. In Bruynooghe, M., ed.: Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers. Volume 3018 of Lecture Notes in Computer Science., Springer-Verlag (2003) 198–213
12. Plump, D.: Confluence of graph transformation revisited. In Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R.C., eds.: Processes, Terms and Cycles. Volume 3838 of Lecture Notes in Computer Science., Springer-Verlag (2005) 280–308
13. Kreowski, H.J., Valiente, G.: Redundancy and subsumption in high-level replacement systems. In: TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations, Springer-Verlag (2000) 215–227

CHRiSM: Chance Rules induce Statistical Models

Jon Sneyers, Wannes Meert, and Joost Vennekens

K.U.Leuven, Belgium
Firstname.Lastname@cs.kuleuven.be

Abstract. A new probabilistic-logic formalism, called CHRiSM, is introduced. CHRiSM is based on a combination of CHR and PRISM. It can be used for high-level rapid prototyping of complex statistical models by means of chance rules. The underlying PRISM system can then be used for several probabilistic inference tasks, including parameter learning. We describe a source-to-source transformation from CHRiSM rules to PRISM, via CHR(PRISM). Finally we discuss the relation between CHRiSM and probabilistic logic programming, in particular, CP-logic.

1 Introduction

Constraint Handling Rules [1, 2] is a high-level language extension based on multi-headed rules. We assume the reader to be familiar with CHR.

The idea of adding probabilities to CHR is not new. In [3], a probabilistic variant of CHR, called PCHR, was introduced. It was implemented by means of a source-to-source transformation [4]. In PCHR, every rule gets a weight which represents its relative probability. A rule is chosen randomly from all applicable rules, according to a probability distribution given by the normalized weights. For example, the following PCHR program implements a fair coin toss:

```
toss <=>0.5: head.
toss <=>0.5: tail.
```

One of the conceptual advantages of PCHR, at least from a theoretical point of view, is that its semantics instantiates the abstract operational semantics ω_t of CHR [2]: every PCHR derivation corresponds to some ω_t derivation.

However, the semantics of PCHR may also lead to some confusion, since it is not so clear what the meaning of the rule weight really is. For example, consider again the above coin tossing example. For the query `toss` we get the answer `head` with 50% chance and otherwise `tail`, so one may be tempted to interpret weights as rule probabilities. However, if the second rule is removed from the program, we will not get the answer `head` with 50% chance, but with a probability of 100%. The reason is that the weights are normalized w.r.t. the sum of the weights of all applicable rules. As a result of this normalization, the actual probability of a rule can only be computed at runtime and by considering the full program. In other words, the probabilistic meaning of a single rule heavily depends on the rest of the PCHR program; there is no localized meaning.

Also, adding weights to propagation rules usually does not make a lot of sense. Consider for example this program:

```
toss ==>0.5: head.
toss ==>0.5: tail.
```

For the above program, the result of the query `toss` will always be `toss, head, tail` (not necessarily in that order): the first propagation rule to fire is chosen randomly, and then the other one has to fire because it is the only applicable rule that is left (so its weight effectively does not matter).

The abstract semantics ω_t of CHR can be instantiated to allow more execution control and more efficient implementations. The best-known example of such an instantiation is the refined semantics ω_r [5]. However, the semantics of PCHR, even though it conforms to ω_t (all PCHR derivations are also ω_t derivations), cannot be instantiated in an analogous way. The reason is that the semantics of PCHR refers to all applicable rules in order to randomly pick one. This conflicts fundamentally with the purpose of instantiations like the refined semantics, which consider only a small portion of the set of applicable rules (i.e., only the rules corresponding to the current active constraint occurrence).

The above issues indicate that perhaps some further investigation of the combination of probabilities and CHR could be useful. In this paper we introduce a different probabilistic variant of CHR, which we call CHRiSM. Its semantics rather differs from that of PCHR and its derivations do not correspond exactly to ω_t derivations (in particular, CHRiSM derivations are partial ω_t derivations). However, the semantics of CHRiSM can be instantiated since it does not refer to the set of all applicable rules. As a result, it can be implemented more efficiently and more execution control can be obtained. Additionally, in the CHRiSM semantics, the rules have a localized meaning: the probabilities do not depend on whether or not other rules are applicable.

CHRiSM can be implemented easily given a CHR(PRISM) system, that is, a CHR system for the host language PRISM. PRISM (PRogramming In Statistical Modeling) is an extension of Prolog with probabilistic built-ins [6]. It has built-in support for several probabilistic inference tasks, including sampling, probability computation, and an expectation-maximization (EM) learning algorithm. These features directly transfer to CHRiSM. These features, in particular EM-learning, are an additional significant advantage of CHRiSM over PCHR: the latter only supports probabilistic execution, i.e. sampling. A state-of-the-art CHR(PRISM) system is currently not available and beyond the scope of this paper. We will however present a prototype implementation of CHRiSM that uses a naive CHR(PRISM) system based on `toychr`.

2 Syntax and Semantics of CHRiSM

CHRiSM extends CHR with two probabilistic statements: it it possible to specify the probability of an entire rule and of the disjuncts in a disjunction.

2.1 Syntax and Informal Semantics

Formally, a CHRiSM program $\mathcal{P}$ consists of a sequence of chance rules. A chance rule is of the following form:

```
P ?? Hk \ Hr <=> G | B.
```

where `P` is a probability expression (as defined below), `Hk` is a conjunction of (kept head) constraints, `Hr` is a conjunction of (removed head) constraints, `G` is a guard condition (a Prolog goal to be satisfied), and `B` is the body of the rule. If `Hk` is empty, the rule is a simplification rule and the backslash is omitted; if `Hr` is empty, the rule is a propagation rule and it is written as "`P ?? Hk ==> ...`".

The meaning of such a chance rule is that, whenever this rule *could* in principle be applied to rewrite its head, this will only happen with a probability given by `P`. Formally, the probability expression `P` is one of the following:

- A number between 0 and 1, indicating the probability that the rule fires. A rule of the form `1 ??` is a normal CHR rule; the "`1 ??`" may be dropped.
- An expression of the form `eval(E)`, where `E` is an arithmetic expression (in Prolog syntax) which may involve variables from the head and guard, which should be fully instantiated when the rule is applicable. The evaluated expression indicates the probability that the rule fires.
- Omitted (so the rule starts with "`??`"): this indicates that the rule probability is unknown.
- A set of variables `V1,...,Vn`: the probability is unknown and it is parametrized in (i.e., the distribution may depend on) the values of `V1,...,Vn`.

Initially, unknown probabilities are set to a uniform distribution (`0.5` in the case of rule probabilities). They can be changed by PRISM's EM-learning algorithm.

Next to these probability assignments to entire rules, CHRiSM also allows to assign probabilities to the disjuncts of a disjunction; that is, the rule body `B` is a conjunction of any of the following:

- Prolog goals and/or CHRiSM constraints (just like in regular CHR);
- CHR$^\vee$-style disjunction [7] (i.e., Prolog disjunction, with backtracking);
- LPAD-style probabilistic disjunction [8] (LPAD: Logic Programs with Annotated Disjunctions), of the form

  ```
  D1:P1 ; ... ; Dn:Pn,
  ```

  where the disjuncts `D`i are rule bodies, and a disjunct `D`i is chosen (committed-choice) with probability `P`i. The probabilities should sum to 1.

In case the probabilities `P`i of the disjuncts are not known, one can also write:

```
P ?? D1 ; ... ; Dn,
```

where `P` can either be omitted (to denote that the probability distribution over the disjuncts is unknown), or a conjunction of variables `V1,..., Vn` (to denote that this distribution is unknown and dependent on the values of the `V`i).

2.2 Operational Semantics

The abstract operational semantics $\omega_t^{??}$ of a CHRiSM program $\mathcal{P}$ is given by a state-transition system that resembles the abstract operational semantics of CHR$^\vee$. In particular, the execution states are defined analogously, except that we additionally define a unique failed execution state which is denoted by "*fail*" (to prevent further rule applications after failure). We refer the reader to [2] for a definition of ω_t execution states. The transitions of $\omega_t^{??}$ are listed in Fig. 1. Note that the first four transitions, together with the last one where `P=1`, correspond to the usual semantics of CHR$^\vee$ (with the minor difference that $\omega_t^{??}$ has an explicit **Fail** transition). We use $\uplus$ for multiset union, and $\bar{\exists}_A B$ to denote $\exists x_1, \ldots, x_n : B$, with $\{x_1, \ldots, x_n\} = vars(B) \setminus vars(A)$, where $vars(A)$ are the (free) variables in A (if A is empty it may be omitted).

1. Fail. $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \mathit{fail}$
where b is a built-in (Prolog) constraint and $\mathcal{D}_\mathcal{H} \models \neg\bar{\exists}(\mathbb{B} \wedge b)$.

2. Solve. $\langle \{b\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$
where b is a built-in (Prolog) constraint and $\mathcal{D}_\mathcal{H} \models \bar{\exists}(\mathbb{B} \wedge b)$.

3. Introduce. $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$
where c is a CHRiSM constraint.

4. Split. $\langle \{d\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \{d_1\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mid \ldots \mid \langle \{d_k\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
where d is a CHR$^\vee$ disjunction of the form d_1 `;` $\ldots$ `;` d_k.

5. Probabilistic Choice. $\langle \{d\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \{d_i\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$
where d is a probabilistic disjunction of the form `P ??` d_1 `;` $\ldots$ `;` d_k or of the form d_1`:`p_1 `;` $\ldots$ `;` d_k`:`p_k. One disjunct d_i is chosen probabilistically according to a distribution parametrized in `P` or given by $p_1, \ldots, p_k$.

6. Maybe-Apply. $\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} X$
where the r-th rule of $\mathcal{P}$ is of the form `P ??` H_1' `\` H_2' `<=> G | B`,
θ is a matching substitution such that $chr(H_1) = \theta(H_1')$ and $chr(H_2) = \theta(H_2')$, $h = (r, id(H_1), id(H_2)) \notin \mathbb{T}$, and $\mathcal{D}_\mathcal{H} \models \mathbb{B} \rightarrow \bar{\exists}_\mathbb{B}(\theta \wedge G)$. With a probability determined by `P`, the resulting state $X = \langle B \uplus \mathbb{G}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$. Otherwise, the resulting state $X = \langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$.

Fig. 1. Transition relation $\rightarrowtail_{\mathcal{P}}$ of the abstract operational semantics $\omega_t^{??}$ of CHRiSM.

In addition to this very nondeterministic abstract operational semantics $\omega_t^{??}$, we can also define more deterministic instantiations of $\omega_t^{??}$, just like ω_r and ω_p are instantiations of ω_t (see also [9]). In the following, we will use a "refined semantics of CHRiSM", defined analogously to [5]. Of course CHRiSM can also be given a "priority semantics", analogous to [10], to get a more intuitive mechanism for execution control.

Any CHRiSM system uses a (computable) execution strategy in the sense of [9]. In the examples in the next section we will assume that this execution strategy is within the strategy class corresponding to the refined semantics of CHRiSM. Note that in [9], an execution strategy completely fixes the derivation for a given input goal. In the context of CHRiSM this is no longer the case because

of the probabilistic choices. However, we may assume that the derivation is fixed if the same choices are made. In other words, the only choice is in the probabilistic choices inside the transitions "**Probabilistic Choice**" and "**Maybe-Apply**"; there is no nondeterminism in choosing which $\omega_t^{??}$ transition to apply next.

The CHRiSM built-in `Q <==> A` denotes that there exist a series of probabilistic choices such that (under the particular execution strategy which is used) a derivation starting with query `Q` results in the answer `A`.

The built-in `Q ===> A` denotes that the answer for query `Q` contains at least `A`: `Q ===> A` holds if `Q <==> B` with `A` $\subseteq$ `B`. In both cases, `Q` and `A` are conjunctions of ground CHRiSM constraints. We also allow "negated" CHRiSM constraints in the right hand side: `Q ===> A,~N` is a shorthand for `Q <==> B` with `A` $\subseteq$ `B` and `N` $\not\subseteq$ `B`.

The following PRISM built-ins can be used when querying a CHRiSM program:

- `sample Q` : probabilistically execute the query `Q`;
- `prob Q <==> A` : compute the probability that `Q <==> A` holds, i.e. the chance that the choices are such that query `Q` results in answer `A`;
- `prob Q ===> A` : compute the probability that the answer for `Q` contains `A`;
- `learn(L)` : perform EM-learning based on a list `L` of observations about derivations (`count/2` can be used to denote multiple identical observations);
- `learn` : perform EM-learning using the facts in a data file whose location can be set using the directive `set_prism_flag(data_source,file(Filename))`.

3 CHRiSM by Example

In this section we illustrate some of the features of CHRiSM by example. As a first toy example, consider the following CHRiSM program for tossing a coin:

```
toss <=> head:0.5 ; tail:0.5.
```

The query "`sample toss`" results in `head` or `tail`, with 50% chance each.

3.1 Rock-paper-scissors

Now consider the following slightly less trivial CHRiSM program to simulate naive "rock-paper-scissors" players:

```
player(P) <=> P ?? rock(P) ; scissors(P) ; paper(P).
rock(P1), scissors(P2) ==> winner(P1).
scissors(P1), paper(P2) ==> winner(P1).
paper(P1), rock(P2) ==> winner(P1).
```

We assume here that each player has his own fixed probability distribution for choosing between rock, scissors, and paper. This is denoted by using `P` as the probability expression for the choice in the first rule: the probability distribution depends on the value of `P` and thus every player has his own distribution.

However, these distributions are not known to us. By default, the unknown probability distributions for, say, `tom` and `jon` are therefore both set to the uniform distribution, which implies, among other things, that each player should win one third of the time. Here is a possible interaction: (user input is in **bold**)

```
| ?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),rock(tom).
| ?- sample player(tom),player(jon)
player(tom),player(jon) <==> rock(jon),paper(tom),winner(tom).
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.333333
```

Now suppose that we watch 100 games, and want to use our observations to obtain a better model of the playing style of both players. If we can fully observe these games, then this is easy: we can just use the frequency with which each player played rock, paper or scissors as an estimate for the probability of him making that particular move. The situation becomes more difficult, however, if the games are only partly observable. For instance, suppose that we do not know which moves the players made, but are only told the final scores: `tom` won 50 games, `jon` won 20, and 30 games were a tie. Deriving estimates for the probabilities of individual moves from this information is less straightforward. For this reason, PRISM comes with a built-in implementation of the EM-algorithm for performing parameter estimation in the presence of missing information [11]. We can use this algorithm to find plausible corresponding distributions:

```
| ?- learn([ count((player(tom),player(jon) ===> winner(tom)),50),
             count((player(tom),player(jon) ===> winner(jon)),20),
count((player(tom),player(jon) ===> ~winner(tom),~winner(jon)),30)])
...
```

The PRISM built-in `show_sw` shows the learned probability distributions, which do indeed correspond (approximately) to the observation frequencies, e.g.:

```
| ?- show_sw
Switch exp1(jon): 1 (p: 0.600570) 2 (p: 0.065368) 3 (p: 0.334061)
Switch exp1(tom): 1 (p: 0.084208) 2 (p: 0.209736) 3 (p: 0.706054)
| ?- prob player(tom),player(jon) ===> winner(tom)
Probability of player(tom),player(jon)===>winner(tom) is: 0.499604
```

3.2 Monopoly dice rolling

The following CHRiSM program implements dice rolling in the Monopoly game. Two dice rolls determine the number of steps to go forward. As long as doubles are rolled, the player may roll again. However, if doubles are rolled three times, the player must go to jail. Note that this program makes use of the refined semantics of CHRiSM: the last rule (roll again) may only be fired if the penultimate rule (go to jail) is not applicable.

```
go <=> roll, roll.
roll <=> ?? die(1);die(2);die(3);die(4);die(5);die(6).
die(X) ==> steps(X).
steps(X), steps(Y) <=> Z is X+Y, steps(Z).
die(X), die(X) <=> again.
again, again, again <=> jail.
again ==> go.
```

We assume that the dice are fair, so the default uniform distribution is correct. If given enough observations, this assumption can be tested by learning the dice probability distribution. If needed we can also distinguish between the two dice:

```
go <=> roll(die_A), roll(die_B).
roll(D) <=> D ?? die(1);die(2);die(3);die(4);die(5);die(6).
```

Here is an example interaction:

```
| ?- sample go
go <==> die(3),die(4),steps(7).
| ?- prob go ===> steps(10)
Probability of go===>steps(10) is: 0.063571
| ?- prob go ===> jail
Probability of go===>jail is: 0.004629
| ?- prob go ===> ∼again, ∼jail
Probability of go===>∼again,∼jail is: 0.833333
```

3.3 Random graphs

Suppose we want to generate a random directed graph, given its nodes. The following rule generates every possible directed edge with probability 50%:

```
0.5 ?? node(A), node(B) ==> edge(A,B).
```

The above rule of course generates dense graphs; if we want to get a sparse graph, say with an average (out-)degree of 3, we can use the following rule. The auxiliary constraint `nb_nodes`(n) contains the total number of nodes n; the probability of the rule is such that each of the $n(n-1)$ possible edges is generated with probability $3/(n-1)$, so we can expect that on average it will generate about $3n$ edges:

```
eval(3/(N-1)) ?? nb_nodes(N), node(A), node(B) ==> edge(A,B).
```

3.4 Bayesian networks

Bayesian networks are one of the most widely used kinds of probabilistic models. A classical example [12] of a Bayesian network is that describing the following alarm system (see Figure 2). Suppose there is some probability that there is a burglary, and also that there is some probability that an earthquake happens.

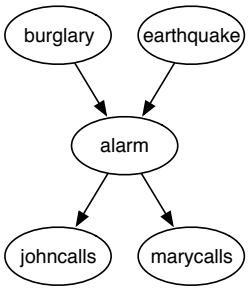

Fig. 2. Bayesian network for alarm system

The probability that the alarm goes off depends on whether those events happen. Also, the probability that John calls the police depends on whether the alarm went off, and similarly for the probability that Mary calls.

This Bayesian network can be described in CHRiSM in a straightforward way:

```
go ==> ?? burglary(yes) ; burglary(no).
go ==> ?? earthquake(yes) ; earthquake(no).
burglary(B), earthquake(E) ==> B,E ?? alarm(yes) ; alarm(no).
A ?? alarm(A) ==> johncalls.
A ?? alarm(A) ==> marycalls.
```

and the probability distributions can be estimated given a number of full observations like

```
go <==> go, burglary(no), earthquake(yes), alarm(yes), marycalls.
go <==> go, burglary(no), earthquake(no), alarm(no).
```

or given partial observations like

```
go ===> johncalls, ~marycalls.
alarm(yes) ===> johncalls.
```

In this way, each Bayesian network can be represented in CHRiSM. We can derive the same information from it as can be derived from the network itself.

4 Prototype Implementation and Future Work

A prototype CHRiSM system was implemented.[1] It is based on a source-to-source transformation to CHR(PRISM) rules. PRISM is implemented on top of B-Prolog, and several advanced CHR systems are currently available for B-Prolog (e.g. [13] and [14]). However, these advanced CHR systems typically make use of non-pure Prolog constructs (e.g. global variables and destructive updates in [13], Action Rules in [14]) which cause some problems in PRISM. Therefore,

[1] Download the CHRiSM system at `http://www.cs.kuleuven.be/~jon/chrism/`

as a simple workaround, the prototype CHRiSM system is based on `toychr`[2], a rather naive implementation of (ground) CHR in pure Prolog, which works fine in PRISM.

4.1 PRISM

PRISM is a probabilistic logic programming language. More precisely: PRISM is Prolog extended with a probabilistic built-in *multi-valued random switch (msw)*. A multi-valued switch atom `msw(term, Result)` represents a probabilistic experiment named `term` (a ground Prolog term), which produces an outcome `Result`. The set of possible outcomes for such an experiment is defined by means of a predicate `values(term,[v1,..., vn])`. By default, a uniform distribution is assumed (all values are equally likely). Different probabilities can be assigned by means of `set_sw(term, [p1, ..., pn])`.

A PRISM program consists out of two parts, rules R and facts F. The facts F define a *base probability distribution* P_F on `msw`-atoms, by means of the `values` and `set_sw` predicates. The rules R are a set of definite clauses, which are allowed to contain the `msw` predicate in the body (but not in the head). This set of clauses R serves to extend the base distribution P to a distribution $P_{DB}(\cdot)$ over the set of Herbrand interpretations: for each interpretation M of the `msw` terms, the probability $P_F(M)$ is assigned to the interpretation I that is the least Herbrand model of $R \cup M$ (*distribution semantics*).

HMM Example. A 2-state HMM is modeled with PRISM as an example. Consider a very simple left-to-right HMM with two states $\{s_0, s_1\}$. s_0 is the initial state and the next state is again s_0 or s_1 which is the end state. In each state, the HMM outputs a symbol either 'a' or 'b'.

```
values(tr(s0), [s0, s1]).
values(out(_), [a,b]).

hmm(Cs) :- hmm(0,s0,Cs).
hmm(T,s1,[C]) :- msw(out(s1),T,C). % Final state
                                   % output symbol and terminate
hmm(T,S,[C|Cs]) :-  S\==s1,        % Not the final state
  msw(out(S),T,C),                 % output symbol
  msw(tr(S),T,Next),               % go to next state
  T1 is T+1,
  hmm(T1,Next,Cs).
```

The first two clauses are declarations to indicate the possible values (the facts). `values`(i, V_i) says that V_i is a list of possible values the switch i can take. The remaining clauses define the probability distribution on the strings generated by the HMM (the rules). `hmm`(Cs) denotes a probabilistic event that the HMM generates a string Cs. `hmm`(T, S, Cs') denotes that the HMM, whose state is S at time T, generates a substring Cs' from that time on.

[2] by Gregory J. Duck, 2004. Download: `http://www.cs.mu.oz.au/~gjd/toychr/`

Alarm System Example. In PRISM the Bayesian network describing the alarm system given before is modeled as follows:

```
values(_,[yes,no]). % all variables are boolean
world(Fire,Earthquake,Alarm,JohnCalls,MaryCalls) :-
   msw(fire,Fire),
   msw(earthquake,Earthquake),
   msw(alarm(Fire,Earthquake),Alarm),
   msw(johncalls(Alarm),JohnCalls),
   msw(marycalls(Alarm),MaryCalls).
```

If the goal is to learn the probabilities from data, this suffices; otherwise, the probabilities can be set explicitly, for example:

```
:- set_sw(johncalls(yes), [0.9, 0.1]).
:- set_sw(johncalls(no),  [0.1, 0.9]).
...
```

4.2 Transformation to CHR(PRISM)

The transformation from CHRiSM to CHR(PRISM) is rather straightforward. We illustrate it by example. Consider again the rule

```
player(P) <=> P ?? rock(P) ; scissors(P) ; paper(P).
```

from Section 3.1. It is translated to the following CHR(PRISM) code:

```
values(experiment1(_),[1,2,3]).
player(P) <=> msw(experiment1(P),X),
              (X=1->rock(P); X=2->scissors(P); X=3->paper(P)).
```

Another example is the random graph rule from Section 3.3:

```
eval(3/(N-1)) ?? nb_nodes(N), node(A), node(B) ==> edge(A,B).
```

which gets translated to the following CHR(PRISM) code:

```
values(experiment1,[1,2]).
nb_nodes(N), node(A), node(B) ==>
     P1 is 3/(N-1), P2 is 1-P1, set_sw(experiment1,[P1,P2]),
     msw(experiment1,X),
     (X=1->edge(A,B); X=2->true).
```

Probabilistic simplification rules and simpagation rules are a bit more tricky since it does not suffice to add a "nop"-disjunct like above. Putting the `msw`-test in the guard of the rule also does not work as expected. In sampling mode, this works fine, but when doing probability computations or learning, an unwanted behavior emerges because of the way PRISM implements explanation search. During explanation search, PRISM essentially redefines `msw`/2 such that

it creates a choice point and tries all values. This causes the guard to always succeed and thus explanations that involve *not* firing a chance rule are erroneously missed. Hence some care has to be taken to translate such rules to PRISM code that behaves correctly. Our solution involves a simple but somewhat ad hoc change to the `toychr` compiler.

In order to further clarify the above issue, consider the PRISM code generated for a CHR rule of the form `Head <=> Guard | Body`. The generated code consists of a series of PRISM clauses, where one clause ends like this:

```
...
( Guard ->
     RemoveConstraints,
     Body
;    Continue ).
```

where `RemoveConstraints` is the code responsible for removing the constraints of `Head` from the constraint store, and `Continue` is the code to try the next applicable rule (in case the guard failed). Clearly, if the `msw`-test is put in the body, we get the problem that the head constraints are removed even if the chance rule was "not applied". If the `msw`-test is put in the guard, the `Continue` branch is never entered when PRISM does explanation search. Instead, we generate code of the following form, which does result in the right behavior:

```
...
( Guard ->
     msw(experimentk,X),
     ( X = 1 ->
          RemoveConstraints,
          Body
     ;    Continue )
; Continue ).
```

4.3 Future work

Our prototype implementation can obviously be improved and extended in many ways. We mention a few general directions for future work:

- The computational performance of `toychr` does not suffice for programs that are not toy examples. It will thus be necessary to get a more advanced CHR system to cooperate with PRISM. One of the obstacles is that PRISM heavily relies on tabling for efficiency [15]; some work has already been done on CHR and tabling in the context of XSB (e.g. [16] and [17]), which may be transferable to PRISM/B-Prolog.
- So far we have only considered ground CHRiSM programs, that is, all constraint arguments are ground at runtime. It is not straightforward to add support for non-ground programs and queries, but it would certainly be useful to have (at least) support for queries like

```
| ?- prob player(tom),player(jon) ===> winner(_)
| ?- prob go ===> steps(S), S > 20
```

– In [3], the notion of probabilistic termination was explored for PCHR programs. Consider for example the Monopoly example. This program always terminates for the query `go`, since the number of re-rolls is at most two. Suppose we remove the "go to jail" rule. Now the program is nonterminating: if we keep rolling doubles, it never ends. However, it is probabilistically terminating since the probability of terminating is 1 (the probability of nontermination is $(1/6)^\infty = 0$). The probability of some derivation, say `prob go ===> steps(20)` can still be computed and learning from a set of observations still makes sense for programs that only terminate probabilistically; however PRISM cannot handle such programs because it will go in an infinite loop during explanation search (or more precisely, run out of stack space). This is not just an issue in CHRiSM but also in PRISM.

5 Relation to Probabilistic Logic Programming

There exist numerous probabilistic extensions of logic programs. One particular family of such extensions is formed by CP-logic or LPADs [18], ProbLog [19], ICL [20], and PRISM itself [6]. All of these can be easily transformed to CHRiSM. Let us consider the case of CP-logic. A theory in this language consists of a set of normal logic programming rules with probabilistically quantified disjunctions in the head; i.e., a CP-logic rule r is of the form:

$$(h_1 : \alpha_1) \vee \cdots \vee (h_n : \alpha_n) \leftarrow \phi$$

with the h_i propositions, the α_i probabilities (with $\sum \alpha_i = 1$) and ϕ a formula. Such a rule expresses that if the formula ϕ is satisfied, a random event will take place, which causes one of the h_i; each α_i represent the probabilities that the associated h_i is the atom that is in fact caused.

We can translate such a rule to CHRiSM by introducing some new symbols. We introduce a symbol $\mathtt{b}_r$ to denote that the body ϕ of this rule r is satisfied, and for each h_i in its head, we introduce a symbol $\mathtt{c}_r^{h_i}$ to represent that h_i is the atom caused by r. For all the symbols s that we now have, we also introduce a symbol `not_`s to represent its negation. We can then translate the above CP-logic rule as:

$$\mathtt{b}_r \texttt{ <=> } \alpha_1 \texttt{:} \mathtt{c}_r^{h_1} \texttt{ ; ... ; } \alpha_n \texttt{:} \mathtt{c}_r^{h_n}$$

and

$$\texttt{not_}\mathtt{b}_r \texttt{ <=> not_}\mathtt{c}_r^{h_1}\texttt{,... , not_}\mathtt{c}_r^{h_n}\texttt{.}$$

In CP-logic, each rule can "fire" at most once. To avoid that the multiset semantics of CHR would give a different result, we add $\mathtt{b}_r$`\ `$\mathtt{b}_r$ `<=> true` to the top of the CHRiSM program.

To relate the $\mathtt{c}_r^{h_i}$ and `not_`$\mathtt{c}_r^{h_j}$, we use: $\mathtt{c}_r^{h_i}$ `==>` `not_`$\mathtt{c}_r^{h_j}$, for all $i \neq j$.

The semantics of CP-logic has a non-monotonic aspect, which stems from its use of "negation-as-failure" as in the well-founded semantics for logic programs. This can be captured in CHRiSM by expressing that a atom h is true if and only if it is caused by at least one rule r_i:

$$\begin{array}{c} \mathtt{c}^h_{r_1} \texttt{ ==> h} \\ \vdots \\ \mathtt{c}^h_{r_n} \texttt{ ==> h} \\ \mathtt{not_c}^h_{r_1}\texttt{,... , }\mathtt{not_c}^h_{r_n} \texttt{ ==> not_h.} \end{array}$$

Here, $r_1, \ldots, r_n$ are all the rules that have h in their head.

All that remains is to define the $\mathtt{b}_r$ and $\mathtt{not_b}_r$ in such a way that they correctly correspond to the truth of the bodies of the rules r. This requires us to encode the logical connectives in CHR. If we push negation down to the atom level, we can simply replace each $\neg h$ with `not_h`. Encoding the meaning of $\vee$ and $\wedge$ is straightforward.

We have now shown that CP-logic can be encoded in CHRiSM in a compact and modular way. It follow that we can do the same for sublogics of CP-logic, such as ProbLog, ICL, and PRISM.

Next to these "logic programming flavored" languages, there also exist a number of formalisms that are inspired by Bayesian networks, such as BLP [21], RBN [22], CLP(BN) [23], and Blog [24]. Based on the encoding of Bayesian networks that we gave in Section 3.4, we can also translate BLPs to CHRiSM. RBNs, CLP(BN) and Blog would be more difficult, because they allow more complex probability distributions, for which CHRiSM currently does not offer support.

As the above paragraphs show, we can encode CP-logic or BLPs using only the `D1:P1; ...; Dn:Pn` construction of CHRiSM. In particular, the ability to assign a probability to an entire rule is not used. This construct offers some expressivity that these other language do not have; namely, it allows to explicitly make things false. For instance, suppose we want to model the evolution of a population. The fact that an individual might die could be represented in CHRiSM as: `0.1 ?? alive <=> true`. In a language such as CP-logic, this would have to be encoding in a roundabout way, using explicit time points and a probabilistic law of persistence:

$$(alive(T+1) : 0.9) \leftarrow alive(T).$$

The CHRiSM representation is more compact and elegant.

6 Conclusion

In this exploratory paper, we have introduced a novel rule-based probabilistic-logic formalism called CHRiSM, which is based on a combination of CHR and PRISM. We have introduced its syntax and semantics and illustrated them with a few example programs. These examples, being only toy examples, show only

a glimpse of the power, elegance, and expressiveness of CHRiSM's chance rules. Just like CHR has important advantages over Prolog (a.o., multi-headedness), CHRiSM has the same advantages over PRISM.

When comparing CHRiSM to PCHR, the authors consider the most important difference to be that CHRiSM has a cleaner and more natural semantics. In contrast to PCHR, the probabilistic meaning of CHRiSM's chance rules is local, that is, it does not depend on the full program and runtime information (the applicability of other rules). This difference between PCHR and CHRiSM can also be approached from the point of view of execution control: while PCHR can only be defined in the framework of a very nondeterministic abstract operational semantics, CHRiSM can also be given a refined operational semantics or a priority semantics.

Another practical advantage of CHRiSM over PCHR is that there are many features inherited from PRISM: the support for computing probabilities, learning from examples, etc. These features essentially come for free — although from the system implementation point of view, the combination of CHR and PRISM also introduces several challenges, most of which have not been tackled yet in this exploratory work.

We described a naive prototype implementation and identified some issues and directions for future work. We also showed how CP-logic can be embedded in CHRiSM. CP-logic (and, by virtue of the embedding of CP-logic in CHRiSM, also CHRiSM) can express many other probabilistic logic formalisms (including ProbLog, ICL, and PRISM).

This paper obviously only scratches the surface of the research questions associated with CHRiSM.

Acknowledgments

Jon Sneyers and Wannes Meert are funded by Ph.D. grants of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Joost Vennekens is a postdoctoral researcher of FWO Vlaanderen.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. Theory and Practice of Logic Programming (2009)
3. Frühwirth, T., Di Pierro, A., Wiklicky, H.: Probabilistic Constraint Handling Rules. In: 11th Intl. Workshop Funct. and (Constraint) Logic Progr. ENTCS 76 (2002)
4. Frühwirth, T., Holzbaur, C.: Source-to-source transformation for a class of expressive rules. In Buccafurri, F., ed.: AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE, Reggio Calabria, Italy (September 2003) 386–397
5. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: ICLP '04. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 90–104

6. Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems **31** (2008)
7. Abdennadher, S., Schütz, H.: CHR$^\vee$, a flexible query language. In Andreasen, T., Christiansen, H., Larsen, H., eds.: FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems. Volume 1495 of LNAI., Roskilde, Denmark, Springer (May 1998) 1–14
8. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In Demoen, B., Lifschitz, V., eds.: ICLP '04. LNCS, vol. 3132, Saint-Malo, France, Springer (2004) 431–445
9. Sneyers, J., Frühwirth, T.: Generalized CHR machines. In Schrijvers, T., Frühwirth, T., Raiser, F., eds.: CHR '08, Hagenberg, Austria (July 2008)
10. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In Leuschel, M., Podelski, A., eds.: 9th International Conference on Principles and Practice of Declarative Programming, Wrocław, Poland, ACM Press (2007) 25–36
11. Kameya, Y., Sato, T.: Efficient EM learning with tabulation for parameterized logic programs. In: Proceedings of the 1st International Conference on Computational Logic (CL2000). Volume 1861 of Lecture Notes in Artificial Intelligence. (2000) 269–294
12. Pearl, J.: Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference. Morgan Kaufmann (1988)
13. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In: 1st Workshop on Constraint Handling Rules. (2004)
14. Schrijvers, T., Zhou, N.F., Demoen, B.: Translating Constraint Handling Rules into Action Rules. In: 3rd Workshop on Constraint Handling Rules. (2006)
15. Zhou, N.F., Sato, T., Shen, Y.D.: Linear tabling strategies and optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
16. Schrijvers, T., Demoen, B., Warren, D.: TCHR: A framework for tabled CHR. Theory and Practice of Logic Programming **8**(4) (2008)
17. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for efficient tabled evaluation. In Hanus, M., ed.: PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages. Volume 4354 of LNCS., Nice, France, Springer (January 2007) 170–184 System's homepage at `http://www.cse.msu.edu/~bss/chr_d`.
18. Vennekens, J., Denecker, M., Bruynooghe, M.: Representing causal information about a probabilistic process. In: Logics in Artificial Intelligence, 10th European Conference, JELIA'06, Proceedings. Volume 4160 of Lecture Notes in Computer Science., Springer (2006) 452–464
19. Raedt, L.D., Kimmig, A., Toivonen, H.: ProbLog: A probabilistic Prolog and its application in link discovery. In: IJCAI. (2007) 2462–2467
20. Poole, D.: The Independent Choice Logic for modelling multiple agents under uncertainty. Artificial Intelligence **94**(1-2) (1997) 7–56
21. Kersting, K., De Raedt, L.: Bayesian logic programming: Theory and tool. In Getoor, L., Taskar, B., eds.: An Introduction to Statistical Relational Learning, MIT Press (2007)
22. Jaeger, M.: Relational Bayesian networks. In: Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97). (1997)
23. Costa, V., Page, D., Cussens, J.: CLP (BN): Constraint logic programming for probabilistic knowledge. Lecture Notes in Computer Science **4911** (2008) 156
24. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D.L., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In Getoor, L., Taskar, B., eds.: Statistical Relational Learning, MIT Press (2007)

A Proposal for a Next Generation of CHR

Peter Van Weert, Leslie De Koninck, and Jon Sneyers

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

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

1 Introduction

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

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

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

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

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

1.1 Constraint Handling Rules

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

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

$$\begin{array}{lllll} \textbf{Simplification} & r \mathbin{@} & H^r & \iff & G \mid B \\ \textbf{Propagation} & r \mathbin{@} H^k & & \implies & G \mid B \\ \textbf{Simpagation} & r \mathbin{@} H^k \setminus H^r & & \iff & G \mid B \end{array}$$

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

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

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

1.2 CHR with Negation as Absence

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

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

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

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

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

1.3 CHR with Rule Priorities

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

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

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

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

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

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

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

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

2 Syntax

2.1 Left-hand Side

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2 Right-hand Side

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

2.3 Priority Constraints

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

Therefore, we propose that rules are no longer assigned numbers explicitly. Instead, each rule is assigned a *rule descriptor*, an arbitrary term that may contain variables occurring in the remainder of the head. This is an extension of the rule names traditionally used in CHR systems (note though that rule descriptors no longer have to be unique). Next, *priority constraints* are specified over these descriptors. Supported constraints are $=$, $<$, $\leq$, $>$ and $\geq$, where 'larger' means 'higher priority'. The operands of priority constraints are *rule*

patterns, or sets thereof, that are matched with rule descriptors either statically or dynamically. A program's priority constraints thus imply a partial preorder on rule instances. We introduce priority constraints by example.

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

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

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

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

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

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

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

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

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

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

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

3 Operational Semantics

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

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

3.1 Rule Normal Form

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

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

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

3.2 Abstract Operational Semantics

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

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

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

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

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

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

Table 1. Transitions of ω_t^2

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

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

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

3.3 Refined Operational Semantics

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

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

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

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

Table 2. Transitions of ω_r^2

The batch function is extended as follows:

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

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

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

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

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

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

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

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

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

4 Annotations

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

4.1 Type and Mode Declarations

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

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

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

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

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

4.2 Constraint Invariants

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

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

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

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

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

4.3 Cardinality Annotations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Conclusion

In this paper, we laid the foundations of CHR*2*, a next generation CHR-based programming language. CHR*2* solves several practical issues with the current

state-of-the-art. We incorporated negation as failure and rule priorities, two very expressive language features, and effectively resolved the disadvantages of earlier proposals. Several novel features further contribute to the expressiveness and practical usability of the language.

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

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

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

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

5.1 Future work

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

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

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

References

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

A Example Derivation

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

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

and priority declaration

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

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

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

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

Fig. 1. Example derivation under ω_r^2